3rd International Conference on Formal Structures for Computation and Deduction

FSCD 2018, July 9–12, 2018, Oxford, United Kingdom

Edited by

Hélène Kirchner

LIPICS

Editor

Hélène Kirchner
Inria
`helene.kirchner@inria.fr`

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

Contents

Invited Talks

Regular Research Papers

System Descriptions and Competition

Contents

◼ **Preface**

The 3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018) was held 9 – 12 July 2018 in Oxford, UK as part of FLoC 2018, 6 – 19 July 2018.

FSCD (`http://fscd-conference.org/`) covers all aspects of formal structures for computation and deduction from theoretical foundations to applications. Initially building on two communities, RTA (Rewriting Techniques and Applications) and TLCA (Typed Lambda Calculi and Applications), FSCD embraces their core topics and broadens their scope to closely related areas in logics and proof theory, new emerging models of computation (e.g. homotopy type theory or quantum computing), semantics and verification in new challenging areas (e.g. blockchain protocols or deep learning algorithms). A special effort was made in 2018 to address some of these new topics through invited speakers. The FSCD program featured four invited talks given by Stéphanie Delaune (CNRS/IRISA, France), Grigore Rosu (U.of Illinois at Urbana-Champaign, US), Peter Selinger (Dalhousie U., Canada), and Valeria Vignudelli (ENS, Lyon, France).

FSCD 2018 received 65 submissions with contributing authors from 21 countries. The program committee consisted of 29 members from 20 countries. Most of the submitted papers were reviewed by at least three PC members (two had only two reviewers) with the help of 112 external reviewers. The reviewing process, which included a rebuttal phase, took place over a period of nine weeks. A total of 27 papers, 26 regular research papers and one system description, were accepted for publication and are included in these proceedings. As an example of tools comparison, the Confluence Competition (CoCo) was presented and took place during the conference.

The Programme Committee awarded the FSCD 2018 Best Paper Award for Junior Researcher to Ambrus Kaposi and András Kovács for their paper "A Syntax for Higher Inductive-Inductive Types". The first author got his PhD less than three years ago and the second is a PhD student, at the time of the Conference.

FSCD 2018 was part of the Federated Logic Conference (FLoC) that brings together several international conferences related to mathematical logic and computer science. FSCD was co-located with SAT, LICS, ITP and CSF, back-to-back with CAV, IJCAR, ICLP and FM.

FSCD 2018 was preceded by the Corrado Bohm Memorial, organised by Mariangiola Dezani, with two invited talks by Henk Barendregt (Radboud University, Nijmegen, The Netherlands) and Silvio Micali (MIT Computer Science & Artificial Intelligence Lab, US).

In addition to the main program, 15 FSCD-associated workshops were planned on three days mostly before the conference:

- 7th International Workshop on Classical Logic and Computation CL&C 2018
- 10th International Workshop on Computing with Terms and Graphs TERMGRAPH 2018
- 7th International Workshop on Confluence IWC 2018
- Higher-Dimensional Rewriting and Algebra HDRA 2018 - 4th edition
- 9th Workshop on Higher Order Rewriting HOR 2018
- Workshop on Homotopy Type Theory and Univalent Foundations HoTT/UF 2018 - 4th edition
- IFIP Working Group 1.6: Rewriting IFIP Meeting 2018 - 20th edition
- 9th Workshop on Intersection Types and Related Systems ITRS 2018

- 2018 Joint Workshop on Linearity & TLLA (5th International Workshop on Linearity and 2nd Workshop on Trends in Linear Logic and Applications)
- International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice LFMTP 2018 - 12th edition
- 7th Workshop on Mathematically Structured Functional Programming MSFP 2018
- Workshop on Modular Knowledge (Tetrapod) - 1st edition
- Programming And Reasoning on Infinite Structures PARIS 2018 - 1st edition
- 5th International Workshop on Rewriting Techniques for Program Transformations and Evaluation WPTE 2018
- 32nd International Workshop on Unification UNIF 2018

This volume of FSCD 2018 is published in the LIPIcs series under a Creative Common license: online access is free to all and authors retain rights over their contributions. We thank in particular Michael Wagner from the Leibniz Center for Informatics at Schloss Dagstuhl for his efficient and reactive support during the production of these proceedings.

Many people helped to make FSCD 2018 a successful meeting. On behalf of the Program Committee, I thank the many authors of submitted papers for considering FSCD as a venue for their work and the invited speakers who have agreed to speak at this meeting. The Program Committee and the external reviewers deserve big thanks for their careful review and evaluation of the submitted papers (the members of the Program Committee and the list of external reviewers can be found in the following pages). The many associated workshops made a big contribution to the lively scientific atmosphere of this meeting and I thank the workshop organizers for their efforts to bring their meetings to Oxford. The EasyChair conference management system was a useful tool in all phases of the work of the Programme Committee. Paula Severi, the Conference Chair for FSCD 2018, deserves warm thanks for producing the Web site, for the smooth functioning of this year's meeting and for coordination with the FLoC organizers. Sandra Alves, as Publicity Chair, made a great contribution in advertising the Conference. The steering committee, lead by Luke Ong, provided valuable guidance in setting up this meeting and is ensuring that FSCD will have a bright and enduring future. Finally, I thank all participants of the conference for creating a lively and interesting event.

FSCD 2018 benefited from being held in-cooperation with the ACM SIGLOG and ACM SIGPLAN.

Hélène Kirchner
Program Chair of FSCD 2018

◼ Steering Committee

T. Altenkirch	Nottingham U.
S. Alves	Porto U.
M. Fernández	King's College London
C. Fuhs	Birkbeck, London U.
D. Kesner	Paris U.
N. Kobayashi	U. Tokyo
D. Miller	Inria
L. Ong (Chair)	Oxford U.
B. Pientka	McGill U.
S. Staton	Oxford U.
R. Thiemann	Innsbruck U.

Program Committee

S. Akshay	IIT Bombay	India
Takahito Aoto	Niigata U.	Japan
Pablo Arrighi	Marseille U.	France
Lars Birkedal	Aarhus U.	Denmark
Eduardo Bonelli	Quilmes U.	Argentina
Adel Bouhoula	Carthage U.	Tunisia
Carlos Castro	Federico Santa María Technical U.	Chile
Ugo Dal Lago	Bologna U.	Italy
Santiago Escobar	U.P. Valencia	Spain
Maribel Fernández	King's College London	UK
Vijay Ganesh	Waterloo U.	Canada
Herman Geuvers	Nijmegen U.	Netherlands
Masahito Hasegawa	Kyoto U.	Japan
Hélène Kirchner (Program Chair)	Inria	France
Paul Blain Levy	U. of Birmingham	UK
Christof Löding	Aachen U.	Germany
Alexandre Miquel	UdelaR, Montevideo	Uruguay
Georg Moser	Innsbruck U.	Austria
Claudia Nalon	Brasilia U.	Brazil
Vivek Nigam	Paraiba U. & fortiss	Brazil & Germany
Peter Csaba Ölveczky	Oslo U.	Norway
Grigore Rosu	Illinois U.	US
Paula Severi (Conference Chair)	U. of Leicester	UK
Viorica Sofronie-Stokkermans	Koblenz-Landau U.	Germany
Nicolas Tabareau	Inria	France
Rene Thiemann	Innsbruck U.	Austria
Alwen Tiu	NTU	Singapore
Femke van Raamsdonk	VU Amsterdam	Netherlands
Lihong Zhi	CAS Bejing	China

External Reviewers

Beniamino Accattoli

María Alpuente

Danil Annekov

Andrei Arusoaie

Kazuyuki Asada

Eugene Asarin

Martin Avanzini

Mauricio Ayala-Rincon

David Baelde

Pablo Barenbaum

Hanene Boussi Rahmouni

Curtis Bright

Arnaud Carayol

David Cerna

Xiaohong Chen

Karl Crary

Deepak D'Souza

Ankush Das

Daniel de Carvalho

Ugo de Liguoro

Andrej Dudenhefner

Alejandro Díaz-Caro

Bertram Felgenhauer

Dan Frumin

Soichiro Fujii

Ronald Garcia

Francesco Antonio Genco

Bernhard Gittenberger

Marek Gluza

Stéphane Graham-Lengrand

Charles Grellois

Amar Hadzihasanovic

Matthew Hague

Jad Hamza

Ichiro Hasuo

Paul He

Hugo Herbelin

Claudio Hermida

Everett Hildenbrandt

Nao Hirokawa

Jan Hoffmann

Ross Horne

Dominic Horsman

Naohiko Hoshino

Florent Jacquemard

Faouzi Jaidi

Michael Jarret

Stefan Kahrs

Jeroen Ketema

Maja Kirkeby

Joachim Kock

Temur Kutsia

Stepan Kuznetsov

Gyesik Lee

Ian Li

John Longley

Giuliano Losa

Dorel Lucanu

Salvador Lucas

Sebastian Maneth

Bassel Mannaa

Alexander Maringele

Simone Martini

Ralph Matthes

Patrick Meredith

Aart Middeldorp

Étienne Miquey

Kenji Miyamoto

Naoki Nishida

David Obwaller

Luc Pellissier

Jean Pichon

Andrew Polonsky

Colin Riba

Christophe Ringeissen

Camilo Rocha

Luca Roversi

David Sabel

Katsuhiko Sano

Julia Sapiña

Haruhiko Sato

Alexis Saurin

Michael Schaper

Aleksy Schubert

Ulrich Schöpp

Andrei Stefanescu

Lutz Strassburger

Sorin Stratulat

Kazushige Terui

Amin Timany

Ashish Tiwari

Alessandro Tosini

Christian Urban

Benno van den Berg

Niels van der Weide

Rob van Glabbeek

Vincent van Oostrom

Daniel Ventura

Pierre Vial

Jamie Vicary

Laurent Vigneron

Renaud Vilmart

Johannes Waldmann

Yuting Wang

Sarah Winkler

Akihisa Yamada

Toshiyuki Yamada

Hans Zantema.

List of Authors

Sandra Alves (5)
DCC-Faculty of Science & CRACS,
University of Porto, Portugal

Nirina Andrianarivelo (6)
LIFO - Université d'Orléans, France

Takahito Aoto (32)
Faculty of Engineering, Niigata University,
Japan

Mauricio Ayala-Rincón (7)
Departments of Mathematics and Computer
Science, Universidade de Brasília, Brazil

Patrick Bahr (8)
IT University of Copenhagen, Denmark

Alexander Baumgartner (9)
Department of Computer Science (DCC),
University of Chile, Santiago, Chile

Gianluigi Bellin (10)
Università di Verona, Italy

Maciej Bendkowski (11)
Jagiellonian University, Faculty of
Mathematics and Computer Science,
Kraków, Poland

Sabine Broda (5)
DCC-Faculty of Science & CMUP,
University of Porto, Portugal

David M. Cerna (12)
Research Institute for Symbolic
Computation, Johannes Kepler University,
Linz, Austria

Łukasz Czajka (13)
University of Copenhagen, Denmark

Giovanni de Felice (17)
Department of Computer Science, University
of Oxford, UK

Stéphanie Delaune (1)
Univ Rennes, CNRS, IRISA, France

Jörg Endrullis (14)
Vrije Universiteit Amsterdam, Department
of Computer Science, Netherlands

Maribel Fernández (7)
Department of Informatics, King's College
London, UK

Simon Forest (15)
LIX, École Polytechnique, France

Thomas Genet (16)
Univ Rennes/Inria/IRISA, France

Amar Hadzihasanovic (17)
RIMS, Kyoto University, Japan

Makoto Hamana (32)
Department of Computer Science, Gunma
University, Japan

Willem B. Heijltjes (10)
University of Bath, UK

Nao Hirokawa (32)
School of Information Science, JAIST, Japan

Mirai Ikebuchi (18)
Massachusetts Institute of Technology,
Cambridge, MA, USA

Rohan Jacob-Rao (19)
Digital Asset, Sydney, Australia

Ambrus Kaposi (20)
Faculty of Informatics, Eötvös Loránd
University, Budapest, Hungary

Jan Willem Klop (14)
Vrije Universiteit Amsterdam, Department
of Computer Science, and Centrum
Wiskunde & Informatica (CWI), Amsterdam,
Netherlands

Christina Kohl (31)
Department of Computer Science, University
of Innsbruck, Austria

András Kovács (20)
Faculty of Informatics, Eötvös Loránd
University, Budapest, Hungary

Temur Kutsia (9, 12)
Research Institute for Symbolic
Computation, Johannes Kepler University
Linz, Austria

Jean-Simon Pacaud Lemay (21)
University of Oxford, Computer Science
Department, UK

Pierre Lescanne (11)
University of Lyon, École normale supérieure
de Lyon, LIP, France

Jordi Levy (9)
Artificial Intelligence Research Institute
(IIIA), Spanish National Research Council,
(CSIC), Barcelona, Spain

Daniel R. Licata (22, 24)
Wesleyan University Dept. Mathematics &
Computer Science, Middletown, USA

Yuya Maeda (26)
Graduate School of Informatics, Nagoya
University, Japan

Bassel Mannaa (23)
Department of Computer Science, IT
University of Copenhagen, Denmark

Aart Middeldorp (30, 31, 32)
Department of Computer Science, University
of Innsbruck, Austria

Samuel Mimram (15)
LIX, École Polytechnique, France

Rasmus Ejlers Møgelberg (23)
Department of Computer Science, IT
University of Copenhagen, Denmark

Julian Nagele (32)
School of Electronic Engineering and
Computer Science, Queen Mary University
of London, UK

Keisuke Nakano (18)
Tohoku University, Sendai, Miyagi, Japan

Daniele Nantes-Sobrinho (7)
Departments of Mathematics and Computer
Science, Universidade de Brasília, Brazil

Max S. New (24)
Northeastern University, Boston, USA

Kang Feng Ng (17)
Department of Computer Science, University
of Oxford, UK

Lê Thành Dũng Nguyễn (25)
Département d'informatique, École normale
supérieure, Paris Sciences et Lettres and
Université Paris 13, Sorbonne Paris Cité,
LIPN, CNRS, France

Naoki Nishida (26, 32)
Graduate School of Informatics, Nagoya
University, Japan

Ian Orton (22)
University of Cambridge, Dept. Computer
Science & Technology, UK

Roy Overbeek (14)
Vrije Universiteit Amsterdam, Department of
Computer Science, Amsterdam, Netherlands

Paweł Parys (27)
University of Warsaw, Poland

Brigitte Pientka (19)
McGill University, Montreal, Canada

Andrew M. Pitts (22)
University of Cambridge, Dept. Computer
Science & Technology, UK

Grigore Rosu (2)
University of Illinois at Urbana-Champaign
and Runtime Verification, Inc., USA

Pierre Réty (6)
LIFO - Université d'Orléans, France

David Sabel (28)
Goethe-University Frankfurt, Germany

Manfred Schmidt-Schauß (28)
Goethe-University Frankfurt, Germany

Peter Selinger (3)
Dalhousie University, Halifax, Canada

Kiraku Shintani (32)
School of Information Science, JAIST, Japan

Matthieu Sozeau (29)
Inria Paris & IRIF, Paris, France

Bas Spitters (22)
Aarhus University Dept. Computer Science,
DK

David Thibodeau (19)
McGill University, Montreal, Canada

Amin Timany (29)
imec-Distrinet, KU Leuven, Belgium

Valeria Vignudelli (4)
Univ Lyon, CNRS, ENS de Lyon, UCB Lyon
1, LIP, France

Mateu Villaret (9)
Departament d'Informàtica, Matemàtica
Aplicada i Estadística, Universitat de Girona,
Spain

Sarah Winkler (30)
Department of Computer Science, University
of Innsbruck, Austria

Harald Zankl (32)
Innsbruck, Austria

Analysing Privacy-Type Properties in Cryptographic Protocols

Stéphanie Delaune

Univ Rennes, CNRS, IRISA, France
stephanie.delaune@irisa.fr

⸻ Abstract ⸻

Cryptographic protocols aim at securing communications over insecure networks such as the Internet, where dishonest users may listen to communications and interfere with them. For example, passports are no more pure paper documents. Instead, they contain a chip that stores additional information such as pictures and fingerprints of their holder. In order to ensure privacy, these chips include a mechanism, i.e. a cryptographic protocol, that does not let the passport disclose private information to external users except the intended terminal. This is just a single example but of course privacy appears in many other contexts such as RFIDs technologies or electronic voting.

Formal methods have been successfully applied to automatically analyse traditional protocols and discover their flaws. Privacy-type security properties (e.g. anonymity, unlinkability, vote secrecy, . . .) are expressed relying on a notion of behavioural equivalence, and are actually more difficult to analyse than confidentiality and authentication properties. We will discuss some recent advances that have been done to analyse automatically equivalence-based security properties, and we will review some issues that remain to be solved in this field.

1 Introduction

Cryptographic protocols are widely used today to secure communications with the aim of achieving various security goals. For instance, TLS (Transport Layer Security) is a protocol that is widely used to provide authentication and encryption in order to send sensitive data such as credit card numbers to a vendor. Those protocols use cryptographic primitives as building blocks such as encryptions, signatures, and hashes.

For a long time, it was believed that designing a strong encryption scheme was sufficient to ensure secure message exchanges. Starting from the 1980's, researchers understood that even with perfect encryption schemes, message exchanges were still not necessarily secure due to some *logical attacks* coming from the poor design of the protocol itself. As an example, we can cite the well-known *man-in-the-middle attack* on the Needham Schroeder protocol [55] that has been discovered by Lowe seventeen years after the publication of the original protocol [53]. This is just a single example for which Lowe proposed a simple fix: the second message

$\{N_a, N_b\}_{\mathsf{pub}(B)}$ is replaced by $\{B, N_a, N_b\}_{\mathsf{pub}(B)}$ in the fixed version of the protocol, i.e. the name of the sender has been simply added into the ciphertext. Such a modification was sufficient to discard the man-in-the-middle attack and to prove the protocol secure. Even if protocols are relatively small programs, they are rather difficult to analyse and the difference between a secure protocol and an insecure one may be rather subtle. For instance, replacing the second message by $\{N_a, N_b, B\}_{\mathsf{pub}(B)}$, i.e. the name of B is now appended at the end of the original message, results in a protocol on which a man-in-the-middle attack similar to the one discovered by Lowe is again possible (see [37] for a description of this attack).

One successful approach when designing and analysing security protocols, is the use of formal methods. The purpose of formal verification is to provide rigorous frameworks and techniques to analyse protocols and find their flaws. For example, a flaw has been discovered in the Single-Sign-On protocol used e.g. by Google Apps. It has been shown that a malicious application could very easily get access to any other application (e.g. Gmail or Google Calendar) of their users [7]. This flaw has been found when analysing the protocol using the Avantssar validation platform [5]. Another example is a flaw on vote secrecy discovered during the formal and manual analysis of an electronic voting protocol [42]. All these results have been obtained using formal symbolic models, where most of the cryptographic details are ignored using abstract structures, and the communication network is assumed to be entirely controlled by an omniscient attacker. Although less precise than computational models used by cryptographers, this symbolic approach benefits from automation and can thus target more complex protocols and scenarios than those analysed using the computational approach. The techniques used in symbolic models have become mature and several tools for protocol verification are nowadays available, e.g. Avantssar platform [5], ProVerif [15], and Tamarin [58].

Most of the results existing in this field focus on reachability properties such as authentication or secrecy: for any execution of the protocol, it should never be the case that an attacker learns some secret (confidentiality property) or that an attacker makes Alice think she's talking to Bob while Bob did not engage a conversation with her (authentication property). However, privacy properties such as vote secrecy, anonymity, or untraceability cannot be expressed as reachability properties. Formally the behaviour of a protocol can be modelled through a process algebra such as the pi-calculus, enriched with terms to model cryptographic messages. Then, privacy-type properties are expressed relying on a notion of behavioural equivalence between processes. For example, Alice's identity remains private if an attacker cannot distinguish a session where Alice is talking from a session where Bob is talking. As mentioned above, many results and tools have been developed in the context of reachability properties. Results for equivalence properties are more rare but a lot of attention has been devoted to its study during the ten past years.

In this paper, we will review existing results and tools dedicated to the study of equivalence-based properties. We will present some recent advances that have been done in this area, and discuss some challenges that remain to be solved.

2   Some examples

We briefly describe in this section some cryptographic protocols on which privacy-type properties are particularly relevant. For illustrative purposes, we first consider a rather simple RFID protocol following a description given in [61] before explaining two protocols coming from the e-passport application: the BAC protocol and its successor the PACE protocol.
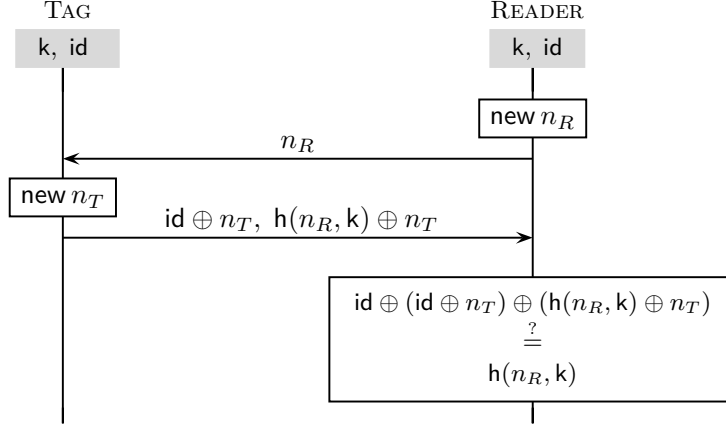
Figure 1 Description of an RFID protocol due to Kim *et al.* [51].

2.1 A simple RFID protocol

To illustrate our formalism along this paper, we will consider a rather simple RFID protocol proposed by Kim et al. [51] in 2007. We follow the description given in [61]. In this protocol, the reader and the tag id share a secret symmetric key k. The protocol does not rely on any encryption algorithm but instead uses a hash function, denoted h, and the exclusive or operator, denoted $\oplus$, which is commutative and associative. Moreover, it has the property that equal terms cancel each other out, i.e. $t \oplus t = 0$ where 0 is the neutral element.

The reader starts by sending a nonce, i.e. a fresh random number n_R. Once it receives this first message, the tag generates its own nonce n_T and computes its answer relying on the hash function and the exclusive-or operator. When receiving this second message, the reader will be able to retrieve n_T from the first component by cancelling out the value id. Then, it will xor this value with the second component and check whether the result is equal to $h(n_R, k)$. Note that since the reader knows n_R and k, it can indeed easily compute the message $h(n_R, k)$.

The aim of this protocol is not only to authenticate the tag but also to ensure its unlinkability. An attacker should not be able to observe whether he has seen the same tag twice or two different tags. Actually, this unlinkability property is not satisfied. An attacker can simply send his own nonce n_R^0 and infer whether the tag in presence is the same or not from the message he received. For this, the attacker simply apply the exclusive-or operator on the two components of the message sent by the tag. The result of this operation is $id \oplus h(n_R^0, k)$ and once n_R^0 is fixed, this value only depends on the identity of the tag.

We will formalise this later on as an indistinguishability property, relying on the notion of trace equivalence.

2.2 Electronic passport

An e-passport is no more a pure paper document but instead contains an RFID chip that stores the critical information printed on the passport. The International Civil Aviation Organisation (ICAO) standard specifies several protocols through which this information can be accessed. In particular, access to the data stored on the passport are protected by the Basic Access Control (BAC) protocol, or now its successor the Password Authenticated Connection Establishment (PACE) protocol.
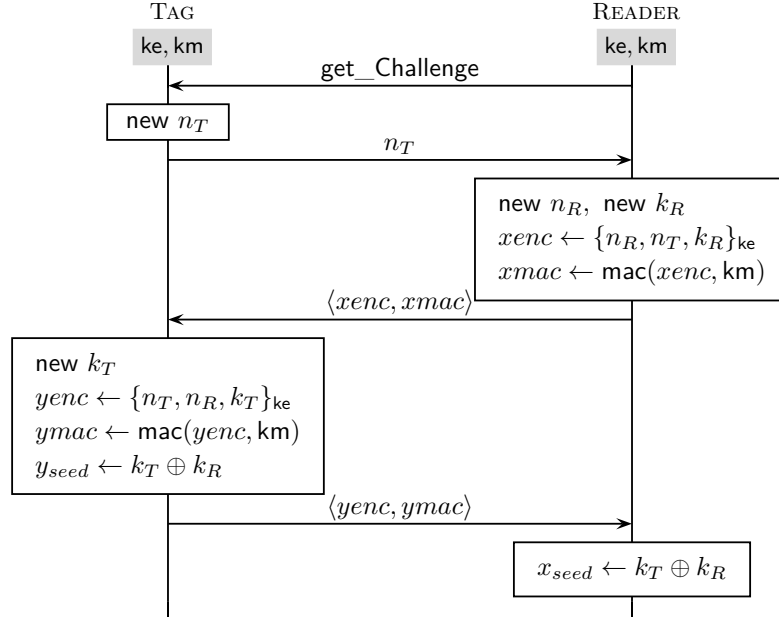
Figure 2 Basic Access Control protocol.

BAC protocol

This is a password-based authenticated key exchange protocol (see Figure 2) whose security relies on two master keys, namely ke and km. Actually, before executing the BAC protocol, the reader optically scans a low entropy secret from which these two keys ke and km are derived. Thus, these keys are symmetric keys shared between the passport (the RFID tag) and the reader. Then, the BAC protocol establishes a key seed from which two sessions keys $kenc$ and $kmac$ are derived. The session keys are then used to prevent skimming and eavesdropping on the subsequent communication with the e-passport. In particular, they are used to encrypt and mac the messages exchanged during the execution of the subsequent protocols.

First, we may note that the nonces n_R and n_T are not placed in the same order in the two ciphertexts: $\{n_R, n_T, k_R\}_{\sf ke}$ and $\{n_T, n_R, k_T\}_{\sf ke}$. Actually, this is not an insignificant choice. This choice prevents a replay attack which would be possible otherwise since it would be possible for an attacker to answer to the message send by a reader without knowing the keys ke and km. Indeed, an attacker could simply replay the message he just received, and this will lead to the computation of the seed $x_{seed} = k_R \oplus k_R = 0$.

Second, the low entropy secret printed in the first page of a passport and from which the keys ke and km are derived makes the BAC protocol vulnerable to off-line guessing attacks. Indeed, an attacker who listens to the communication will learn e.g. $\{n_R, n_T, k_R\}_{\sf ke}$. Then, he can simply try to decrypt this ciphertext using all possible values for ke until he finds a value that allows him to obtain n_T (nonce that has been sent in clear and that is therefore known by the attacker).

Third, we may note that when the passport receives an incorrect message $\langle xenc, xmac \rangle$, the behaviour of the passport is not specified, Due to this, some implementations of the BAC protocol breaches unlinkability [29]: in the french implementation, the passport tag replies different error messages depending on whether the problem comes from an incorrect
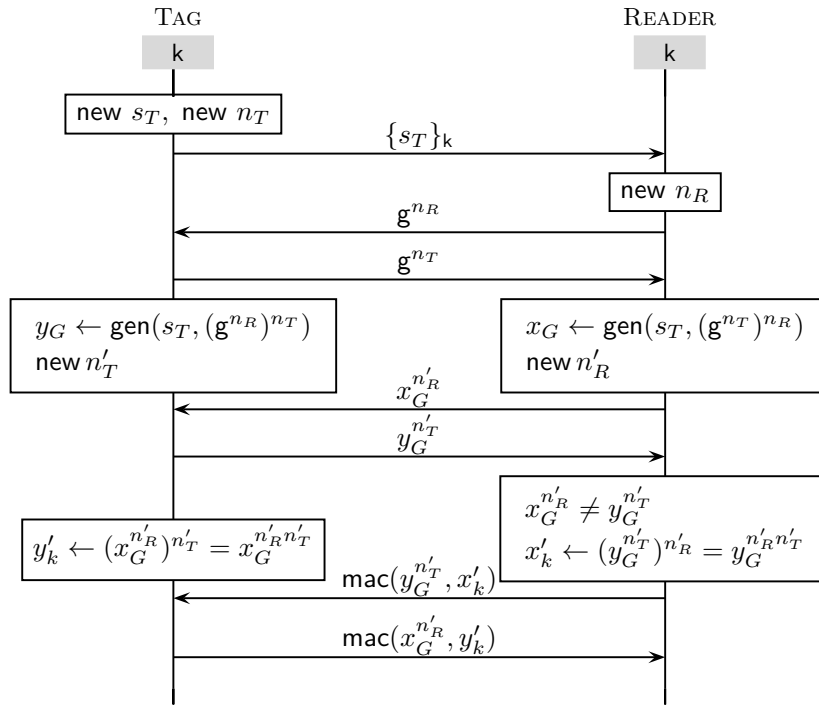
Figure 3 Password Authenticated Connection Establishment protocol.

mac or an incorrect nonce (i.e. the nonce n_T inside the ciphertext is not the one previously generated by the passport). An attacker could then trace a passport (without knowing the keys ke and km) in the following way:

1. he listens to a first session between a reader and a tag T and store $m = \langle xenc, xmac \rangle$;
2. then, in a different session, he sends the message m and wait for the tag's response;
   a. if he receives a nonce error then he knows that the tag succeeded to mac $xenc$ with his own key ke and so this tag is T;
   b. if he receives a mac error then he knows that the tag is not T.

This gives the attacker a way to distinguish between two different passports. Such a flaw does not exist in other implementations where the same error messages is sent in both cases.

PACE protocol

The Password Authenticated Connection Establishment protocol [56] (PACE) has been proposed by the Bundesamt für Sicherheit in der Informationstechnik (BSI) to replace the BAC protocol. Similarly to BAC, the purpose of PACE is to establish a secure channel based on an optically-scanned key k. A description is given in Figure 3. The tag and the reader perform a first Diffie-Hellman exchange and derive G. Then, they perform a second Diffie-Hellman exchange based on the parameter G computed at the previous step, and they derive a session key k'. In a final stage, they confirm the values that have been exchanged using message authentication codes.

First, we may note that the low entropy of the secret k is not a problem anymore assuming that the decryption operation on the ciphertext $\{s_T\}_k$ will not fail when the key used to decrypt is not k. This means that the resulting computation $\mathsf{sdec}(\{s_T\}_k, k_I)$ will be a valid message even if $k \neq k_I$, and thus the protocol will pursue its execution normally.

Second, we would like to comment on the disequality test performed by the reader. Such a test is important to prevent an attacker to execute with success the PACE protocol. Without such a test, an attacker can eavesdrop a message $\{s_T\}_k$ from an honest session, and then reuse it to execute a session with a reader. He simply has to send the ciphertext, and then answer to the reader by replaying the message he just received. This means that the attacker would not have to know k to successfully execute the protocol whereas he is supposed to know it to compute G. Of course, this directly leads to an authentication issue that can be turned into a linkability attack.

Third, the fact that the format of the two last messages are similar is surprising. Due to this, an attacker can send $\{s_T\}_k$ (eavesdrop during a previous session) to two different readers and then simply forward the messages from one reader to another. Both readers will be able to compute the two rounds of Diffie-Hellman, and the mac verification phase will not prevent this behaviour. Even if in practice, this flaw seems hard to exploit, it could be a real privacy concern in some other contexts. Actually, as proposed in [48], this flaw can be fixed by adding tags in the two last messages in order to avoid confusions between reader's messages and tag's messages.

3   Modelling protocols

Several symbolic models have been proposed for cryptographic protocols. The first one has been described by Dolev and Yao [45] and several other models have been proposed since then (e.g. strand spaces [59], multiset rewriting [19], spi-calculus [3]). A unified model would enable better comparisons between the different existing results but unfortunately such a model does not exist currently. Nevertheless, all existing models share some common features: messages are modelled using first-order terms, and they propose some constructions for modelling communication and taking into account the concurrency nature of these programs.

3.1   Messages as terms

In symbolic models, messages are a key concept. Whereas messages are bitstrings in the real-world (and in the computational approach as well), they are modelled using first-order terms within the symbolic model. Formally, we consider an infinite set $\mathcal{N}$ of *names* to represent atomic data such as keys, nonces, and we also consider two infinite sets of *variables* $\mathcal{X}$ and $\mathcal{W}$. The variables in $\mathcal{X}$ are used to model unknown parts of the messages expected by a participant, whereas variables in $\mathcal{W}$, called *handles*, are used as pointers. They refer to messages that have been previously sent on the network and that are therefore known to the attacker.

To model cryptographic primitives, such as encryptions, signatures, hashes, *etc*, we rely on function symbols, namely a *signature*, that allows one to build terms representing messages sent over the network by the participants. The set of terms built from a set of atomic data A by applying function symbols in a signature Σ are denoted $\mathcal{T}(\Sigma, A)$.

▶ **Example 1.** To model the BAC and the RFID protocols described in Section 2, we may consider the signature:

$$\Sigma_{\mathsf{ex}} = \{\mathsf{senc}, \mathsf{sdec}, \langle\,\rangle, \mathsf{proj}_1, \mathsf{proj}_2, \mathsf{mac}, \mathsf{h}, \oplus, 0\}.$$

The function symbols senc and sdec (both of arity 2) represent symmetric encryption, whereas $\langle\,\rangle$ (arity 2) is used to concatenate messages. The two components of such an operator can be retrieved using the projection functions proj_1 and proj_2 (both of arity 1). We also consider

a function symbol to model an hash function, the symbol h (arity 1), as well as a function symbol mac (arity 2) to model message authentication codes. Lastly, the function symbol $\oplus$ (arity 2) is used to model the exclusive-or operator, and the constant 0 is its neutral element.

Then, we assign a meaning to the function symbols through an equational theory. Formally, we consider a set of equations between terms (without names), and we denote $=_E$ the smallest congruence relation which is closed under the substitution of terms for variables.

▶ **Example 2.** Going back to our previous example, we will typically consider the following set E_{ex} of equations:

$$
\begin{aligned}
\mathsf{proj}_1(\langle x, y \rangle) &= x & x \oplus (y \oplus z) &= (x \oplus y) \oplus z & x \oplus 0 &= x \\
\mathsf{proj}_2(\langle x, y \rangle) &= y & x \oplus y &= y \oplus x & x \oplus x &= 0 \\
\mathsf{sdec}(\mathsf{senc}(x, y), y) &= x
\end{aligned}
$$

Considering $m = \mathsf{senc}(\langle n_R, \langle n_T, k_R \rangle \rangle, \mathsf{ke})$, we have that $\mathsf{proj}_1(\mathsf{proj}_2(\mathsf{sdec}(m, \mathsf{ke}))) =_{E_{ex}} n_T$. We may note that the symbols mac and h are not involved in any equation. Those primitives are modelled using free function symbols since they are one-way functions which are typically assumed to be collision resistant.

Sometimes, function symbols are split into two sets: *constructors* and *destructors*. In such a case, a rewriting system is used to assign a meaning to the function symbols. Constructors symbols, typically senc, $\langle \ \rangle$, etc are used to build messages, whereas destructor symbols, such as sdec, proj_1, and proj_2, are only there to perform computations meaning that a rewriting rule has to apply to make them disappear. If no rewriting rule applies, and the destructor is still there, it means that the computation fails, and the resulting term is not considered as a message. This gives us two slightly different ways to model e.g. symmetric encryption. Both are useful when modelling protocols depending on the properties of the encryption scheme. For instance, going back to the PACE protocol, it is important here to model encryption relying on an equation to take into account the fact that $\mathsf{sdec}(\mathsf{senc}(s_T, \mathsf{k}), \mathsf{k}')$ is a computation that does not lead to a failure but instead gives a result, i.e. a message, and the reader will proceed with the resulting value.

At a particular point in time, while engaging in one or more sessions of one or more protocols, an attacker may know a sequence of messages (i.e. terms without variable) $u_1, \ldots, u_n$. This means that he knows all messages and also their order. When analysing equivalence-based security properties, it is not enough to say that the attacker knows the set of terms $\{u_1, \ldots, u_n\}$. In the applied-pi calculus [2], such a sequence of messages is organised into a *frame*, i.e. a substitution of the form:

$$\phi = \{w_1 \mapsto u_1, \ldots, w_n \mapsto u_n\}.$$

The handle $w_i \in \mathcal{W}$ enables us to refer to the message u_i, and these variables will allow us to make explicit the order in which these messages are sent. Given a frame ϕ, we denote $dom(\phi)$ its *domain*, i.e. $dom(\phi) = \{w_1, \ldots, w_n\}$.

We need also to consider computations feasible by an attacker. We call such a computation a *recipe*. Formally, a recipe is a term built from (public) function symbols and handles from $\mathcal{W}$.

▶ **Definition 3.** Given a frame ϕ and a term $u \in \mathcal{T}(\Sigma, \mathcal{N})$, we say that u is *deducible* from ϕ, denoted $\phi \vdash_E u$, when there exists a recipe R, i.e. a term in $\mathcal{T}(\Sigma, dom(\phi))$, such that $R\phi =_E u$.

3.2 Protocols as processes

A popular way to model protocols is to rely on a process algebra. Several calculus have been proposed to model protocols, e.g. CSP [49], spi-calculus [3], applied-pi calculus [2]. They have similar constructs as those in the pi-calculus introduced by Milner in 1999 [54]. However, instead of exchanging atomic data, terms that are exchanged are first-order terms. This allows us to model in a more faithful way cryptographic protocols that use cryptographic messages. Typically, considering a set $\mathcal{C}h$ of public channel names, processes are generated by a grammar as follows:

$$
\begin{array}{llll}
P, Q & := & 0 & \text{null} \\
 & | & P \mid Q & \text{parallel} \\
 & | & \mathsf{in}(c, x).P & \text{input} \\
 & | & \mathsf{out}(c, u).P & \text{output} \\
 & | & !P & \text{replication} \\
 & | & \mathsf{new}\, n.P & \text{restriction} \\
 & | & \mathsf{if}\ u_1 = u_2\ \mathsf{then}\ P\ \mathsf{else}\ Q & \text{conditional}
\end{array}
$$

where $u_1, u_2, u \in \mathcal{T}(\Sigma, \mathcal{N} \cup \mathcal{X})$, $c \in \mathcal{C}h$, and $n \in \mathcal{N}$.

Most of the constructions are rather standard in process algebra. As usual, the null process, denoted 0, represents the process that does nothing. Such a process is often omitted for sake of conciseness. The process $P \mid Q$ runs P and Q in parallel meaning that we do not know in which order the actions of P and Q will be done. All the interleavings should be considered. The process $\mathsf{in}(c, x).P$ waits to receive a message on the public channel c, and then continues as indicated in P. However, the occurrence of the variable x in P will be replaced by the received message. The process $\mathsf{out}(c, u).P$ outputs the message u on the public channel c, and then continues as P. The process $!P$ represents an infinite number of copies of P running in parallel, i.e. $P \mid \ldots \mid P$. The restriction $\mathsf{new}\, n.P$ is used to model the creation in a process of new random numbers (e.g., nonces or key material). The process $\mathsf{if}\ u_1 = u_2\ \mathsf{then}\ P\ \mathsf{else}\ Q$ first checks whether u_1 is equal to u_2 (modulo the equational theory), and runs P if equality holds or runs Q otherwise. Note that the terms u, u_1, and u_2 that occur in the grammar may contain variables. However, these variables will be instantiated during the execution, and these terms will become ground when the evaluation will take place.

The constructions $\mathsf{new}\, n.P$ and $\mathsf{in}(c, x).P$ are binding constructs, respectively for the name n and for the variable x, and in both cases the scope of the binding is P.

▶ **Example 4.** To illustrate our syntax, we consider the RFID protocol described in Section 2.1. Using our formalism, the two roles of this protocol are modelled as follows:

$$
\begin{aligned}
P_{\mathsf{tag}} &= \mathsf{in}(c_T, x).\ \mathsf{new}\, n_T.\ \mathsf{out}(c_T, \langle \mathsf{id} \oplus n_T,\ \mathsf{h}(\langle x, k \rangle) \oplus n_T \rangle).\ 0 \\
P_{\mathsf{reader}} &= \mathsf{new}\, n_R.\ \mathsf{out}(c_R, n_R).\ \mathsf{in}(c_R, y).\ \mathsf{if}\ (\mathsf{proj}_1(y) \oplus \mathsf{id}) \oplus \mathsf{proj}_2(y) \\
&= \mathsf{h}(\langle n_R, , k \rangle)\ \mathsf{then}\, 0\, \mathsf{else}\, 0.
\end{aligned}
$$

where $c_T, c_R \in \mathcal{C}h$, $\mathsf{id} \in \mathcal{N}$, and $x, y \in \mathcal{X}$.

Then, we may consider the process $\mathsf{new}\, k.\, \mathsf{new}\, \mathsf{id}.(P_{\mathsf{tag}} \mid P_{\mathsf{reader}})$ which corresponds to one instance of each role. We may also consider more complex scenario. For instance, the process $\mathsf{new}\, k.\, \mathsf{new}\, \mathsf{id}.!\, (P_{\mathsf{tag}} \mid P_{\mathsf{reader}})$ represents multiple instances of the tag id (with key k) and multiple instances of a reader ready to communicate with tag id. Lastly, the process $!\, \mathsf{new}\, k.\, \mathsf{new}\, \mathsf{id}.!\, (P_{\mathsf{tag}} \mid P_{\mathsf{reader}})$ represents a situation with many tags (and readers), each of them being able to execute many instances of the protocol.

$$\begin{array}{lll}
\textsc{Then} & (\{\texttt{if } u_1 = u_2 \texttt{ then } P_1 \texttt{ else } P_2\} \uplus \mathcal{P}; \phi) \xrightarrow{\tau} (P_1 \uplus \mathcal{P}; \phi) & \text{when } u_1 =_\mathsf{E} u_2 \\[4pt]
\textsc{Else} & (\{\texttt{if } u_1 = u_2 \texttt{ then } P_1 \texttt{ else } P_2\} \uplus \mathcal{P}; \phi) \xrightarrow{\tau} (P_2 \uplus \mathcal{P}; \phi) & \text{when } u_1 \neq_\mathsf{E} u_2 \\[4pt]
\textsc{In} & (\{\mathsf{in}(c, z).P\} \uplus \mathcal{P}; \phi) \xrightarrow{\mathsf{in}(c,R)} (P\{z \mapsto u\} \uplus \mathcal{P}; \phi) & \text{when } \phi \vdash_\mathsf{E} u \\[4pt]
\textsc{Out} & (\{\mathsf{out}(c, u).P\} \uplus \mathcal{P}; \phi) \xrightarrow{\mathsf{out}(c,w)} (P \uplus \mathcal{P}; \phi \cup \{\mathsf{w}_{i+1} \mapsto u\}) & \text{where } i = |dom(\phi)|. \\[4pt]
\textsc{New} & (\{\mathsf{new }\, n.P\} \uplus \mathcal{P}; \phi) \xrightarrow{\tau} (P\{n \mapsto n'\} \uplus \mathcal{P}; \phi) & \text{where } n' \in \mathcal{N} \text{ is fresh} \\[4pt]
\textsc{Par} & (\{P_1 \mid P_2\} \uplus \mathcal{P}; \phi) \xrightarrow{\tau} (\{P_1, P_2\} \uplus \mathcal{P}; \phi) & \\[4pt]
\textsc{Repl} & (\{!P\} \uplus \mathcal{P}; \phi) \xrightarrow{\tau} (\{!P, P\} \uplus \mathcal{P}; \phi) &
\end{array}$$

Figure 4 Semantics of our processes.

Configurations represent processes together with a frame representing the knowledge of the attacker so far.

▶ **Definition 5.** A *configuration* is a pair $(\mathcal{P}; \phi)$ where
- $\mathcal{P}$ is a multiset of *ground processes*; and
- $\phi = \{\mathsf{w}_1 \mapsto u_1, \ldots, \mathsf{w}_n \mapsto u_n\}$ is a *frame*.

The applied-pi calculus, as introduced in [2], does not consider this notion of configurations but rely instead on a notion of extended processes and a notion of structural equivalence to identify processes that are equal modulo e.g. associativity and commutativity of the parallel operator. Our notion of configurations, also used in some other works, can be seen as a more canonical way to represent an extended process.

Then, we define the operational semantics of our calculus through a labelled transition system over configurations explaining how a process can evolve (see Figure 4). All the rules are rather standard and correspond to the informal semantics introduced at the beginning of this section. For instance, when outputting a message u on the public channel c, the resulting message is stored into the frame ϕ and is given to the attacker through the handle w_i. The rule IN is more involved. The idea is that the attacker can build any term using his current knowledge and then send the resulting message to the participant. Therefore, the participant is ready to accept any term deducible by the attacker form ϕ. The rules THEN and ELSE allow one to execute a conditional. Note that the equality is done modulo E. The three remaining rules allow one to execute a restriction, split a parallel composition, and unfold a replication. The IN and OUT rules are the only observable actions.

▶ **Example 6.** To continue with our running example, we consider $\mathcal{P}_\mathsf{same} = \{P_\mathsf{tag}, P_\mathsf{tag}\}$, and $\mathcal{P}_\mathsf{diff} = \{P_\mathsf{tag}, P'_\mathsf{tag}\}$ where P'_tag is as P_tag but id and k have been replaced by id' and k'. Let $\phi_0 = \{\mathsf{w}_1 \mapsto \mathsf{id}, \mathsf{w}_2 \mapsto \mathsf{id}'\}$. We have that:

$$\begin{array}{ll}
& (\{P_\mathsf{diff}\}; \phi_0) \\[4pt]
\xrightarrow{\mathsf{in}(c_T, n_R^0)} \xrightarrow{\tau} & (\{\mathsf{out}(c_T, \langle \mathsf{id} \oplus n_T, \mathsf{h}(\langle n_R^0, \mathsf{k}\rangle) \oplus n_T\rangle), P'_\mathsf{tag}\}; \phi_0) \\[4pt]
\xrightarrow{\mathsf{out}(c_T, \mathsf{w}_3)} & (\{P'_\mathsf{tag}\}; \phi_0 \uplus \{\mathsf{w}_3 \mapsto \langle \mathsf{id} \oplus n_T, \mathsf{h}(\langle n_R^0, \mathsf{k}\rangle) \oplus n_T\rangle\}) \\[4pt]
\xrightarrow{\mathsf{in}(c_T, n_R^0)} \xrightarrow{\tau} & (\{\mathsf{out}(c_T, \langle \mathsf{id}' \oplus n_T', \mathsf{h}(\langle n_R^0, \mathsf{k}'\rangle) \oplus n_T'\rangle).0\}, \phi_0 \uplus \{\mathsf{w}_3 \mapsto \langle \mathsf{id} \oplus n_T, \mathsf{h}(\langle n_R^0, \mathsf{k}\rangle) \oplus n_T\rangle\}) \\[4pt]
\xrightarrow{\mathsf{out}(c_T, \mathsf{w}_4)} & (\{0\}, \phi_0 \uplus \{\mathsf{w}_3 \mapsto \langle \mathsf{id} \oplus n_T, \mathsf{h}(\langle n_R^0, \mathsf{k}\rangle) \oplus n_T\rangle, \; \mathsf{w}_4 \mapsto \langle \mathsf{id}' \oplus n_T', \mathsf{h}(\langle n_R^0, \mathsf{k}'\rangle) \oplus n_T'\rangle\})
\end{array}$$

We denote ϕ_diff the resulting frame. The same sequence of actions is also feasible starting from $(\mathcal{P}_\mathsf{same}; \phi_0)$. Indeed, we have that:

$$(\mathcal{P}_{\mathsf{same}}; \phi_0) \xrightarrow{\mathsf{in}(c_T, n_R^0) \cdot \tau \cdot \mathsf{out}(c_T, \mathsf{w}_3) \cdot \mathsf{in}(c_T, n_R^0) \cdot \tau \cdot \mathsf{out}(c_T, \mathsf{w}_4)} (0; \phi_{\mathsf{same}})$$

where $\phi_{\mathsf{same}} = \phi_0 \uplus \{\mathsf{w}_3 \mapsto \langle \mathsf{id} \oplus n_T, \mathsf{h}(\langle n_R^0, \mathsf{k} \rangle) \oplus n_T \rangle, \ \mathsf{w}_4 \mapsto \langle \mathsf{id} \oplus n_T', \mathsf{h}(\langle n_R^0, \mathsf{k} \rangle) \oplus n_T' \rangle\}$.

4   Modelling privacy-type properties

In order to express privacy-type properties such as the unlinkability property briefly discussed in Section 2, we need to formally define the notion of indistinguishability we are interested in. The notion of *trace equivalence* is formally introduced in Section 4.1, and then we explain how to express privacy-type security properties such as vote-privacy, unlinkability, or strong flavours of secrecy relying on this notion.

4.1   Trace equivalence

To start, we consider two protocols P and Q, and we assume a passive attacker who simply listens to the communication. We would like to know whether such a passive attacker is able (by simply listening to the communication) to tell which protocol is currently under execution: i.e. P or Q. Typically, the attacker will observe two sequences of messages, i.e. two frames, and he will try to distinguish them. Intuitively, two frames ϕ and ψ are in static equivalence if an attacker cannot distinguish them, i.e. any test that holds in ϕ also holds in ψ.

▶ **Definition 7.** Two frames ϕ and ψ are in *static equivalence*, written $\phi \sim_\mathsf{E} \psi$, if they have the same domain, i.e. $dom(\phi) = dom(\psi)$, and for any recipes $R, R' \in \mathcal{T}(\Sigma, dom(\phi))$, we have that: $R\phi =_\mathsf{E} R'\phi$ if, and only if, $R\psi =_\mathsf{E} R'\psi$.

▶ **Example 8.** Consider the two following frames:
- $\phi_{\mathsf{diff}} = \phi_0 \uplus \{\mathsf{w}_3 \mapsto \langle \mathsf{id} \oplus n_T, \mathsf{h}(\langle n_R^0, \mathsf{k} \rangle) \oplus n_T \rangle, \mathsf{w}_4 \mapsto \langle \mathsf{id}' \oplus n_T', \mathsf{h}(\langle n_R^0, \mathsf{k}' \rangle) \oplus n_T' \rangle\}$, and
- $\phi_{\mathsf{same}} = \phi_0 \uplus \{\mathsf{w}_3 \mapsto \langle \mathsf{id} \oplus n_T, \mathsf{h}(\langle n_R^0, \mathsf{k} \rangle) \oplus n_T \rangle, \mathsf{w}_4 \mapsto \langle \mathsf{id} \oplus n_T', \mathsf{h}(\langle n_R^0, \mathsf{k} \rangle) \oplus n_T' \rangle\}$.

The following test holds in ϕ_{same}: $\mathsf{proj}_1(\mathsf{w}_3) \oplus \mathsf{proj}_2(\mathsf{w}_3) \overset{?}{=} \mathsf{proj}_1(\mathsf{w}_4) \oplus \mathsf{proj}_2(\mathsf{w}_4)$.

Indeed, we have that:
- $[\mathsf{proj}_1(\mathsf{w}_3) \oplus \mathsf{proj}_2(\mathsf{w}_4)]\phi_{\mathsf{same}} =_{\mathsf{E}_{\mathsf{ex}}} (\mathsf{id} \oplus n_T) \oplus (\mathsf{h}(\langle n_R^0, \mathsf{k} \rangle) \oplus n_T) =_{\mathsf{E}_{\mathsf{ex}}} \mathsf{id} \oplus \mathsf{h}(\langle n_R^0, \mathsf{k} \rangle)$, and
- $[\mathsf{proj}_1(\mathsf{w}_3) \oplus \mathsf{proj}_2(\mathsf{w}_4)]\phi_{\mathsf{same}} =_{\mathsf{E}_{\mathsf{ex}}} (\mathsf{id} \oplus n_T') \oplus (\mathsf{h}(\langle n_R^0, \mathsf{k} \rangle) \oplus n_T') =_{\mathsf{E}_{\mathsf{ex}}} \mathsf{id} \oplus \mathsf{h}(\langle n_R^0, \mathsf{k} \rangle)$.

However, this test does not hold in ϕ_{diff} since $\mathsf{id} \neq \mathsf{id}'$ and $\mathsf{k} \neq \mathsf{k}'$. This means that an attacker can observe a difference between these two frames by xoring the two components of each message and checking whether this computation yields an equality, therefore retrieving the attack described in Section 2.1. Note that such an equality crucially relies on the algebraic properties of the exclusive-or operator.

Then, trace equivalence is the active counterpart of static equivalence taking into account the fact that the attacker may interfere during the execution of the process in order to distinguish between the two situations. Given a configuration $K = (\mathcal{P}; \phi)$, we define $\mathsf{trace}(K)$ as follows: $\mathsf{trace}(K) = \{(\mathsf{tr}, \phi') \mid (\mathcal{P}; \phi) \xrightarrow{\mathsf{tr}} (\mathcal{P}'; \phi')$ for some configuration $(\mathcal{P}'; \phi')\}$.

▶ **Definition 9.** Given two configurations K_P and K_Q, $K_P \sqsubseteq_t K_Q$ if for every $(\mathsf{tr}, \phi) \in \mathsf{trace}(K_P)$, there exists $(\mathsf{tr}', \psi) \in \mathsf{trace}(K_Q)$ such that tr and tr' are equal up to τ actions, and $\phi \sim_\mathsf{E} \psi$. The configuration K_P and K_Q are in *trace equivalence*, denoted $K_P \approx_t K_Q$, if $K_P \sqsubseteq_t K_Q$ and $K_Q \sqsubseteq_t K_P$.

4.2 Some security properties

We show here how the notion of trace equivalence can be used to model interesting privacy-type properties.

Unlinkability

Intuitively, protocols are said to provide unlinkability (or untraceability) according to the ISO/IEC 15408-2 standard, if they

> [...] ensure that a user may make multiple uses of [them] without others being able to link these uses together.

Formally, this is often defined as the fact that an attacker should not be able to distinguish a scenario in which the same agent (i.e. the user) is involved in many sessions from one that involved different agents in each session. Going back to our BAC protocol, this can be expressed through the following equivalence:

$$! \text{ new ke. new km. } ! (P_{\mathsf{tag}}(\mathsf{ke}, \mathsf{km}) \mid P_{\mathsf{reader}}(\mathsf{ke}, \mathsf{km}))$$
$$\approx_t$$
$$! \text{ new ke. new km.} (P_{\mathsf{tag}}(\mathsf{ke}, \mathsf{km}) \mid P_{\mathsf{reader}}(\mathsf{ke}, \mathsf{km})).$$

In case of the french implementation of the BAC protocol, as explained in Section 2.2, it has been shown that this equivalence does not hold [4].

▶ **Example 10.** To illustrate our notion of trace equivalence, we consider the RFID protocol given in Section 2.1. In order to simplify the setting, we consider a simple scenario which consists of two sessions of the role of the tag. We assume that one session is executed by the tag with identity id and key k, whereas the second one is executed either by the same tag or another one. We would like to know whether the attacker is able to distinguish these two situations. This corresponds to the configurations $K_{\mathsf{same}} = (\mathcal{P}_{\mathsf{same}}; \phi_0)$ and $K_{\mathsf{diff}} = (\mathcal{P}_{\mathsf{diff}}; \phi_0)$ described in Example 6. Actually, we have that K_{same} and K_{diff} are not in trace equivalence. More precisely, we have that $K_{\mathsf{same}} \not\sqsubseteq_t K_{\mathsf{diff}}$. Indeed, we have shown that:

- $K_{\mathsf{same}} \xrightarrow{\mathsf{in}(c_T, n_R^0) \cdot \tau \cdot \mathsf{out}(c, \mathsf{w}_3) \cdot \mathsf{in}(c_T, n_R^0) \cdot \tau \cdot \mathsf{out}(c_T, \mathsf{w}_4)} (0, \phi_{\mathsf{same}})$ (see Example 6); and
- $[\mathsf{proj}_1(\mathsf{w}_3) \oplus \mathsf{proj}_2(\mathsf{w}_3) =_{\mathsf{E}_{\mathsf{ex}}} \mathsf{proj}_1(\mathsf{w}_4) \oplus \mathsf{proj}_2(\mathsf{w}_4)]\phi_{\mathsf{same}}$ (see Example 8).

However, the only configuration $(\mathcal{P}', \phi')$ such that (up to some τ actions)

$$K_{\mathsf{diff}} \xrightarrow{\mathsf{in}(c, n_R^0) \cdot \mathsf{out}(c, \mathsf{w}_3) \cdot \mathsf{in}(c, n_R^0) \cdot \mathsf{out}(c, \mathsf{w}_4)} (P', \phi')$$

is $(0, \phi_{\mathsf{diff}})$ and we have seen that $\mathsf{proj}_1(\mathsf{w}_3) \oplus \mathsf{proj}_2(\mathsf{w}_3) \overset{?}{=} \mathsf{proj}_1(\mathsf{w}_4) \oplus \mathsf{proj}_2(\mathsf{w}_4)$ does not hold in ϕ_{diff} (see Example 8).

This corresponds to the attack scenario briefly described in Section 2.1.

Vote secrecy

In the context of electronic voting, privacy means that the vote of a particular voter is not revealed to anyone. This is one of the fundamental security properties that an electronic voting system has to satisfy.

Vote secrecy is typically defined (see e.g. [44]) by the fact that an observer should not observe when two honest voters swap their votes, i.e. distinguish between a situation where

Alice votes *yes* and Bob votes *no* and a situation where these two voters have swapped their vote. This security property is formally expressed as follows:

$$S[V(A, yes) \mid V(B, no)] \approx_t S[V(A, no) \mid V(B, yes)].$$

Ideally, such an equivalence should be established considering an empty context S. However, very often such a property holds under some trusted assumptions. For instance, we may have to trust the tallying authority. The context S makes explicit all these assumptions.

Strong flavours of secrecy

The notion of secrecy usually considered by symbolic approaches is a weak form of secrecy expressed as a non-deducibility property. However, relying on trace equivalence, we can also express strong forms of secrecy. Intuitively, strong secrecy means that an attacker cannot see any difference when the value of the secret changes [14]. One way to express this it to let the attacker choose the values of the secret:

$$\mathsf{in}(c, x_1).\mathsf{in}(c, x_2).P(x_1) \approx_t \mathsf{in}(c, x_1).\mathsf{in}(c, x_2).P(x_2).$$

Intuitively, in the equivalence above, $P(x)$ is the protocol in which the value of the secret is replaced by x, i.e. by a value chosen by the attacker. Another flavour of secrecy of interest is *real-or-random secrecy*. The idea is to let the attacker interact with the protocol, and once the secret value has been established, typically a fresh session key k, we want to see if the attacker is able to distinguish the real situation (the one in which the fresh established key k will be used) from an ideal situation in which the key k is replaced by fresh random value r. If the adversary is unable to distinguish these two scenarios, we say that the protocol satisfies real-or-random secrecy of the secret key.

   The notion of trace equivalence can also be used in presence of low entropy secret such as the values ke and km in the BAC protocol to check resistance of the protocol to off-line guessing attacks. This can be modelled relying on trace equivalence by checking whether the attacker can see the difference between a scenario where the real password is revealed at the end, and another one where a wrong password is revealed (see e.g. [36]).

5 Verifying equivalence-based properties

The formal symbolic approach allows one to benefit from existing verification tool that rely on various techniques ranging from model-checking to resolution, and rewriting techniques. This is appealing as manual proofs are tedious and error-prone. However, verifying in such a setting the most simple form of secrecy (expressed as a non-deducibility of a term) is a difficult problem which is well-known to be undecidable. Privacy-type properties are actually more difficult to handle and have been shown undecidable even for some classes where secrecy is actually decidable [32].

5.1 Warm-up

Several papers are devoted to the study of the intruder deduction problem, i.e. the problem of deciding whether a term (typically the secret) is deducible from a given set of terms representing the knowledge of the attacker. This problem has been shown decidable (often in PTIME) for various equational theories, e.g. homomorphic encryption, blind signatures, various equational theories with an associative and commutative symbol (AC). However,

Theory E	Deduction	Static Equivalence
subterm convergent	PTIME [1]	
blind sign., addition, homo. encryption	decidable [1]	
ACU	NP-complete	decidable [1] PTIME [39]
ACUN/AG	PTIME [27]	PTIME [1, 39]
ACUh	NP-complete [52]	decidable [39]
ACUNh/AGh	PTIME [43]	decidable [39]
$AGh_1 \ldots h_n$	decidable [39]	decidable [39]

◾ **Figure 5** Some decidability and complexity results for deduction and static equivalence.

when considering equivalence-based properties, the natural question we have to solve is not to decide whether a term is deducible or not, but rather whether two frames are indistinguishable or not. This problem can be formally stated as follows:

Static equivalence problem:

Input Two frames ϕ and ψ having the same domain.

Output Are ϕ and ψ in static equivalence, i.e. $\phi \sim_E \psi$?

Again depending on the equational theory under study, this problem may or may not be decidable. Actually, even if such a problem is often more complex to solve than the intruder deduction problem, this problem is now well-understood. Efficient (often PTIME) algorithms and tools (e.g. FAST [35], and KISS [33]) have been developed to solve this problem for various equational theories. Some existing results for deduction and static equivalence are briefly summarised in Figure 5. Moreover, thanks to the combination result provided in [39], deduction and static equivalence are also decidable for the union of any disjoint theories of this tabular.

5.2 Bounded number of sessions

When analysing a protocol, a reasonable assumption under which the verification problem has been shown decidable is the so-called bounded number of sessions assumption. This amounts to consider processes without replication. Note that processes without replication allows us to consider traces of bounded length, but the problem remains difficult: the labelled transition system representing all the possible interactions of the honest participants with the attacker is still infinitely branching. This issue has been tackled in various ways using forms of symbolic execution and the development of dedicated procedures. Obtaining a symbolic semantics to avoid potentially infinite branching of execution trees due to inputs from the environment is often a first step towards automation of equivalence.

Some theoretical results. Under such an assumption, the problem of deciding trace equivalence has been first shown decidable in [50], where a fragment of the spi-calculus (no replication, no else branch) is considered. In 2005, Baudet designs a constraint solving procedure that is not only able to solve satisfiability problems (sufficient for reachability

properties) but also to establish equivalences (i.e. two systems have the same sets of solutions), which are needed when one wants to verify equivalence-based security properties [13]. Few years later, a shorter proof of this result was proposed by Chevalier and Rusinowitch in [28]. In this work, it is shown that when two processes are not in trace equivalence, then there exists a small witness of this fact. The main issue with the results mentioned so far is practicality. Consequently, they have not been implemented.

Since then, a lot of progress has been made leading to more efficient procedures implemented in various verification tools. We review the most popular ones and briefly explain their features.

Spec.   This tools implements a decision procedure for the notion of *open bisimulation*: a notion that is strictly stronger than trace equivalence [60]. Processes given in input are written in the spi-calculus syntax and else branches are not allowed. Regarding the cryptographic primitives, the tool has been recently extended to deal with asymmetric primitives, and therefore is now able to handle all the standard primitives.

Akiss.   The procedure described in [20] deals with rich user-defined term algebras provided that they can be defined using a convergent rewriting system enjoying the *finite variant property* [34].This includes all the standard primitives, but also some other primitives such blind signatures, and trapdoor commitment used e.g. in electronic voting protocols. However, due to some approximations, this procedure is only able to check trace equivalence for the class of determinate processes. Moreover, its termination is only guaranteed for subterm convergent equational theories. Regarding the input syntax, processes are written as linear roles and originally the tool only allows inputs, outputs, and equality tests. Recently, some extensions have been implemented. In particular, the procedure has been extended to deal with the exclusive-or operator [8], and various RFID protocols have been analysed such as the RFID protocol presented in Section 2.1. The tool has also been extended to deal with else branches [47].

Apte/DeepSec.   The tool Apte [21] implements the decision procedure described in [23]. Such a procedure deals with all the standard cryptographic primitives. Actually, the procedure presented in [23] allows for slightly more general processes than those presented in Section 3 since it deals with private channels and internal communications. This procedure has been extended to deal with some forms of *side-channel* attacks regarding the length of messages [25], and the computation time [24]. Recently, getting some inspiration from the Apte tool, a new verification tool DeepSec has been implemented [26]. It deals with a large variety of cryptographic primitives that encompasses all the standard primitives. Moreover, it is significantly more efficient than other existing tools, namely Spec, Akiss, and its predecessor Apte.

SAT-Equiv.   Following an approach originally developed for checking reachability properties [6], SAT-Equiv relies on more general verification techniques, namely graph planning and SAT-solving [38]. The procedure deals with symmetric encryption and pairs, and only consider simple processes (each process in parallel works on a dedicated channel) without else branches. However, an extension is currently under preparation and the tool will be able to cover all standard primitives soon. In order to benefit from graph planning and SAT-solvers, the size of messages has to be bounded and this bound needs to be practical. The soundness of the tool is based on a typing result [30] guaranteeing the existence of a

witness of non-equivalence where messages comply to a certain format (induced by a typing system). The resulting implementation, SAT-Equiv, outperforms other existing tools. It can analyse several sessions (typically more than 20 for rather simple protocols) where most existing tools have to stop after few sessions. Termination has been established, and this is the most efficient tool able to decide trace equivalence. However, the class of processes that it is able to consider is rather limited (e.g. no else branch, simple processes satisfying a type-compliance assumption).

5.3 Unbounded number of sessions

The decidability results and the tools mentioned so far only consider a bounded number of sessions, and thus assume that the protocol is executed a limited number of times. The problem is that even if the protocol has been proved secure for n sessions, there is no guarantee that the protocol will remain secure if it is executed one more time.

Some theoretical results. It is well-known that replication allowing us to model an unbounded number of sessions leads to undecidability even when considering a simple secrecy property. The first decidability result for trace equivalence has been obtained for a rather restricted class: the class of *ping-pong protocols* built using standard primitives (but without pairs) [32]. This result has been obtained through a characterisation of equivalence of protocols in terms of equality of languages of (generalised, real-time) deterministic pushdown automata.

Assuming finitely many nonces and keys, another decidability result has been obtained in [30] for the class of simple processes built using symmetric encryptions and pairs. Such a decidability result is based on a typing result which means that messages comply to a certain format. A well-known class of protocols that satisfies such a requirement, is the class of *tagged protocols*. The typing result mentioned above has also been used to establish the first decidability result in presence of fresh nonces [31]. This decidability result inherits the restrictions of the typing result (symmetric encryption only, type-compliance) on which it is based. Additionally, a notion of dependency graph allowing one to represent the dependencies between the actions of the protocols is carefully designed. In case this graph is acyclic, a bound on the length of an attack trace can be deduced, giving us an algorithm to decide trace equivalence.

ProVerif, Tamarin, and Maude-NPA. Since the problem of checking trace equivalence for rich class of protocols is undecidable, many works aim at developing procedures that are sound but not complete w.r.t. trace equivalence. In particular, several tools consider the notion of *diff-equivalence* (a notion stronger than trace equivalence). This notion has been introduced in [16] and implemented in the ProVerif tool. Since then, this notion of diff-equivalence has been integrated in Tamarin [12] and Maude-NPA [57]. Due to the fact that the equivalence under study is the so-called notion of diff-equivalence, these tools are not well-suited to analyse some privacy-type properties such as unlinkability, or vote secrecy.

To extend the scope of the ProVerif tool, several extensions have been recently proposed to go beyond diff-equivalence, e.g. [22, 17]. For instance, ProSwapper [17] allows one to go beyond diff-equivalence by rearranging automatically processes before launching ProVerif. This front end is particularly relevant to analyse vote secrecy. ProVerif has also been used as a back end to analyse anonymity and unlinkability properties [48]. This approach proposes sufficient conditions that are actually checkable using ProVerif, and from which the security of the protocol can be established. This method allows to automatically verify unlinkability and

anonymity of some protocols that were out of the scope of existing tools, e.g. unlinkability of the fixed version of the BAC protocol has been established for the first time relying on this technique, and some of the weaknesses presented in Section 2.2 on the PACE protocol have been discovered using this method.

Whereas ProVerif and Maude-NPA are completely automatic, Tamarin provides two ways of constructing proofs: an efficient, fully automated mode that uses heuristics to guide proof search and an interactive mode. Regarding the cryptographic primitives, these tools support a rich term algebra including all the standard primitives. In addition, Tamarin also supports Diffie-Hellman exponentiation, and recently exclusive-or has been added into the tool. The Maude-NPA tool also supports a rich term algebra but the tool suffers from termination issues, especially when considering the exclusive-or operator. Despite some non-termination issues that may happen from time to time, these tools are efficient. For instance, ProVerif generally concludes within few seconds. These good performances are due to some well-chosen over-approximations that are done on the protocols at the beginning of the security analysis that may lead sometimes to false attacks.

Type-Eq.   Recently, an approach based on type systems has been developed and implemented in the tool Type-Eq [40, 41]. This approach is very efficient and can prove security of protocols that require a mix of bounded and unbounded number of sessions This tool only consider the standard cryptographic primitives, and requires the user to enter all the information regarding types. While this approach allows to go beyond diff-equivalence, e.g. allowing else branches to be matched with then branches, it is not yet possible to analyse e.g. unlinkability of the BAC protocol.

6   Some challenges

In the past ten years, equivalence-based properties have received a lot of attention and we now have tools to check automatically privacy-type security properties when considering rather simple protocols. However, new applications are coming or are already there (electronic voting, contactless payment, keyless systems, . . .) and these applications often rely on security protocols that can not be analysed relying on existing verification tools due to various reasons.

State-explosion problem.   Systems we are interested in are highly concurrent and all the existing methods and tools which naively explore all possible symbolic interleavings are causing the so called state-explosion problem. This problem seriously limits the practical impact of tools such as Akiss, Spec, Apte and, to a lesser extent, DeepSec. Actually, recent works [9, 10] have partially addressed this issue by developing dedicated partial order reduction (POR) techniques to dramatically reduce the number of interleavings to explore. They have been implemented in Apte, Akiss, and DeepSec, and brought significant speed-up. However, these techniques can only be applied on action-deterministic processes, and this is not sufficient to analyse e.g. unlinkability. To mitigate this problem, it should be possible to leverage classical POR techniques for use in the specific security setting. A recent result in this direction has been obtained in [11] regarding the persistent and sleep sets techniques, and other POR techniques deserve attention to obtain performance gains.

Cryptographic primitives.   Most of the tools deal with standard primitives, i.e. encryptions, signatures, and hashes. However, many protocols, such as RFID protocols or electronic voting protocols rely on primitives that do not fall into this class. For instance, protocols

that contain some time-critical steps often rely on low-level operator to reduce computation (and communication) time. As demonstrated by some recent works (e.g. [8]), dealing with a simple operator such as the exclusive-or and its algebraic properties in the symbolic setting is actually challenging. Electronic voting protocols have to achieve antagonist security properties and they often rely on exotic cryptographic primitives to try to achieve them, e.g. blind signatures, zero-knowledge proofs, homomorphic encryptions, . . . To avoid missing attacks, these primitives together with their algebraic properties have to be modelled faithfully when performing the formal security analysis.

Mutable global states. Many modern protocols involve a notion of state meaning that some data are conveyed from one session to another through e.g. a register. This is the case for instance of many RFID protocols as those presented in [18, 61]. Several protocols proposed by the 3rd Generation Partnership Project (3GPP) as a standard for 3G and 4G mobile network communications are also stateful. For instance, the Authentication and Key Agreement (AKA) protocol relies on a state to store a counter across different sessions, and a state is also used in a crucial way to store temporary identifiers (namely TMSI) in the TMSI reallocation procedure. Moreover, these protocols are supposed to guarantee unlinkability. Existing results regarding the formal verification of such protocols model states in a very abstract way, considering for instance that the client and the server already (magically) share a fresh name instead of modelling the sequence number mechanism. Recent advances have been made in this direction though an extension of the Tamarin prover [46]. Nevertheless, methods for checking trace equivalence on stateful protocols are in their infancy.

References

1 M. Abadi and V. Cortier. Deciding knowledge in security protocols under equational theories. *Theoretical Computer Science*, 387(1-2):2–32, 2006.

2 M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proc. 28th Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115. ACM Press, 2001.

3 Martín Abadi and Andrew D Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proc. of the 4th ACM conference on Computer and communications security*, pages 36–47. ACM, 1997.

4 Myrto Arapinis, Tom Chothia, Eike Ritter, and Mark Ryan. Analysing unlinkability and anonymity using the applied pi calculus. In *Proc. 23rd Computer Security Foundations Symposium (CSF'10)*, pages 107–121. IEEE Computer Society Press, 2010.

5 A. Armando et al. The AVANTSSAR platform for the automated validation of trust and security of service-oriented architectures. In *Proc. 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12)*, volume 7214, pages 267–282. Springer, 2012.

6 Alessandro Armando, Roberto Carbone, and Luca Compagna. SATMC: a SAT-based model checker for security-critical systems. In *Proc. 20th international Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14)*, pages 31–45. Springer, 2014. `doi:10.1007/978-3-642-54862-8_3`.

7 Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuellar, and Llanos Tobarra Abad. Formal analysis of saml 2.0 web browser single sign-on: Breaking the saml-based single sign-on for google apps. In *Proc. 6th ACM Workshop on Formal Methods in Security Engineering (FMSE 2008)*, pages 1–10, 2008.

8 David Baelde, Stéphanie Delaune, Ivan Gazeau, and Steve Kremer. Symbolic verification of privacy-type properties for security protocols with xor. In *Proc. 30th IEEE Computer*

Security Foundations Symposium (CSF'17), pages 234–248. IEEE Computer Society Press, 2017.

9   David Baelde, Stéphanie Delaune, and Lucca Hirschi. Partial order reduction for security protocols. In *Proc. 26th International Conference on Concurrency Theory (CONCUR'15)*, volume 42 of *LIPIcs*, pages 497–510. Leibniz-Zentrum für Informatik, 2015.

10 David Baelde, Stéphanie Delaune, and Lucca Hirschi. A reduced semantics for deciding trace equivalence. *Logical Methods in Computer Science*, 2017.

11 David Baelde, Stéphanie Delaune, and Lucca Hirschi. POR for Security Protocols Equivalences - Beyond Action-Determinism. *arXiv*, 2018. `arXiv:1804.03650`.

12 David Basin, Jannik Dreier, and Ralf Sasse. Automated symbolic proofs of observational equivalence. In *Proc. 22nd Conference on Computer and Communications Security (CCS'15)*, pages 1144–1155. ACM, 2015.

13 Mathieu Baudet. Deciding security of protocols against off-line guessing attacks. In *Proc. 12th ACM conference on Computer and communications security (CCS'05)*, pages 16–25. ACM, 2005.

14 Bruno Blanchet. Automatic proof of strong secrecy for security protocols. In *Proc. . 2004 Symposium on Security and Privacy*, pages 86–100. IEEE Computer Society Press, 2004.

15 Bruno Blanchet. An automatic security protocol verifier based on resolution theorem proving (invited tutorial). In *Proc. 20th International Conference on Automated Deduction (CADE-20)*, July 2005.

16 Bruno Blanchet, Martín Abadi, and Cédric Fournet. Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 75(1):3–51, 2008.

17 Bruno Blanchet and Ben Smyth. Automated reasoning for equivalences in the applied pi calculus with barriers. In *Proc. 29th Computer Security Foundations Symposium (CSF'16)*, 2016.

18 Mayla Brusó, Konstantinos Chatzikokolakis, and Jerry Den Hartog. Formal verification of privacy for RFID systems. In *Proc. 23rd Computer Security Foundations Symposium (CSF'10)*, pages 75–88. IEEE Computer Society Press, 2010.

19 I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *Proc. 12th Computer Security Foundations Workshop (CSFW'99)*, pages 55–69. IEEE Computer Society Press, 1999.

20 Rohit Chadha, Ştefan Ciobâcă, and Steve Kremer. Automated verification of equivalence properties of cryptographic protocols. In *Proc. European Symposium on Programming (ESOP'12)*, pages 108–127. Springer, 2012.

21 Vincent Cheval. Apte: an algorithm for proving trace equivalence. In *Proc. 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14)*, volume 8413 of *LNCS*, pages 587–592, 2014.

22 Vincent Cheval and Bruno Blanchet. Proving more observational equivalences with ProVerif. In *Proc. 2nd Conference on Principles of Security and Trust (POST'13)*, volume 7796 of *LNCS*, pages 226–246. Springer, 2013.

23 Vincent Cheval, Hubert Comon-Lundh, and Stéphanie Delaune. Trace equivalence decision: Negative tests and non-determinism. In *Proc. 18th ACM Conference on Computer and Communications Security (CCS'11)*, pages 321–330. ACM Press, 2011.

24 Vincent Cheval and Véronique Cortier. Timing attacks in security protocols: symbolic framework and proof techniques. In *Proc. 4th Conference on Principles of Security and Trust (POST'15)*, pages 280–299. Springer, 2015.

25 Vincent Cheval, Véronique Cortier, and Antoine Plet. Lengths may break privacy – or how to check for equivalences with length. In *Proc. 25th International Conference on*

Computer Aided Verification (CAV'13), volume 8044 of *LNCS*, pages 708–723. Springer Berlin Heidelberg, 2013.

26 Vincent Cheval, Steve Kremer, and Itsaka Rakotonirina. Deepsec: Deciding equivalence properties in security protocols - theory and practice. In *Proc. 39th IEEE Symposium on Security and Privacy (S&P'18)*. IEEE Computer Society Press, 2018. Accepted for publication.

27 Y. Chevalier, R. Küsters, M. Rusinowitch, and M. Turuani. Deciding the security of protocols with Diffie-Hellman exponentiation and product in exponents. In *Proc. 23rd Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS'03)*, volume 2914 of *LNCS*, pages 124–135. Springer-Verlag, 2003.

28 Yannick Chevalier and Michael Rusinowitch. Decidability of symbolic equivalence of derivations. *Journal of Automated Reasoning*, 48(2):263–292, 2012.

29 Tom Chothia and Vitaliy Smirnov. A traceability attack against e-passports. In *Proc. 14th International Conference on Financial Cryptography and Data Security (FC'10)*, 2010.

30 Rémy Chrétien, Véronique Cortier, and Stéphanie Delaune. Typing messages for free in security protocols: the case of equivalence properties. In *Proc. 25th International Conference on Concurrency Theory (CONCUR'14)*, volume 8704 of *LNCS*, pages 372–386. Springer, 2014.

31 Rémy Chrétien, Véronique Cortier, and Stéphanie Delaune. Decidability of trace equivalence for protocols with nonces. In *Proc. 28th Computer Security Foundations Symposium (CSF'15)*, pages 170–184. IEEE Computer Society Press, 2015.

32 Rémy Chrétien, Véronique Cortier, and Stéphanie Delaune. From security protocols to pushdown automata. *ACM Transactions on Computational Logic*, 17(1:3), 2015.

33 Ştefan Ciobâcă, Stéphanie Delaune, and Steve Kremer. Computing knowledge in security protocols under convergent equational theories. *Journal of Automated Reasoning*, 48(2):219–262, 2012.

34 Hubert Comon-Lundh and Stéphanie Delaune. The finite variant property: How to get rid of some algebraic properties. In *Proc. International Conference on Rewriting Techniques and Applications (RTA'05)*, pages 294–307. Springer, 2005.

35 Bruno Conchinha, David A. Basin, and Carlos Caleiro. FAST: an efficient decision procedure for deduction and static equivalence. In *Proc. 22nd International Conference on Rewriting Techniques and Applications, RTA 2011*, volume 10 of *LIPIcs*, pages 11–20. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

36 Ricardo Corin, Jeroen Doumen, and Sandro Etalle. Analysing password protocol security against off-line dictionary attacks. *Electonic Notes in Theoretical Computer Science*, 121:47–63, 2005.

37 Véronique Cortier. *Vérification automatique des protocoles cryptographiques*. Thèse de doctorat (PhD thesis), Laboratoire Spécification et Vérification, ENS Cachan, France, mar 2003.

38 Véronique Cortier, Antoine Dallon, and Stéphanie Delaune. Sat-equiv: an efficient tool for equivalence properties. In *Proc. 30th IEEE Computer Security Foundations Symposium (CSF'17)*. IEEE Computer Society Press, aug 2017.

39 Véronique Cortier and Stéphanie Delaune. Decidability and combination results for two notions of knowledge in security protocols. *Journal of Automated Reasoning*, 48(4):441–487, 2012.

40 Véronique Cortier, Niklas Grimm, Joseph Lallemand, and Matteo Maffei. A type system for privacy properties. In *24th ACM Conference on Computer and Communications Security (CCS'17)*, pages 409–423. ACM, October 2017.

41 Véronique Cortier, Niklas Grimm, Joseph Lallemand, and Matteo Maffei. Equivalence properties by typing in cryptographic branching protocols. In *Proc. 7th International Conference on Principles of Security and Trust (POST'18)*, pages 160–187, April 2018.

42 Véronique Cortier and Ben Smyth. Attacking and fixing helios: An analysis of ballot secrecy. In *Proc. 24th Computer Security Foundations Symposium (CSF'11)*, pages 297–311. IEEE Computer Society Press, 2011.

43 Stéphanie Delaune. Easy intruder deduction problems with homomorphisms. *Information Processing Letters*, 97(6):213–218, 2006. URL: `http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/SD-ipl05.pdf`.

44 Stéphanie Delaune, Steve Kremer, and Mark D. Ryan. Symbolic bisimulation for the applied pi calculus. *Journal of Computer Security*, 18(2):317–377, mar 2010.

45 D. Dolev and A. C. Yao. On the security of public key protocols. In *Proc. 22nd Symposium on Foundations of Computer Science (FCS'81)*, pages 350–357. IEEE Computer Society Press, 1981.

46 Jannik Dreier, Lucca Hirschi, Saša Radomirovic, and Sasse Ralf. Automated unbounded verification of stateful cryptographic protocols with exclusive OR operations. In *Proc. 31st IEEE Computer Security Foundations Symposium (CSF'18)*. IEEE Computer Society Press, 2018.

47 Ivan Gazeau and Steve Kremer. Automated analysis of equivalence properties for security protocols using else branches. In *Proc. 22nd European Symposium on Research in Computer Security (ESORICS'17)*, volume 10493 of *Lecture Notes in Computer Science*, pages 1–20. Springer, sep 2017. `doi:10.1007/978-3-319-66399-9_1`.

48 Lucca Hirschi, David Baelde, and Stéphanie Delaune. A method for verifying privacy-type properties: the unbounded case. In *Proc. 37th Symposium on Security and Privacy (S&P'16)*, 2016.

49 C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978. `doi:10.1145/359576.359585`.

50 Hans Hüttel. Deciding framed bisimilarity. *Electronic Notes in Theoretical Computer Science*, 68(6):1–18, 2003.

51 Il-Jung Kim, Eun Young Choi, and Dong Hoon Lee. Secure mobile RFID system against privacy and security problems. In *Third International Workshop on Security, Privacy and Trust in Pervasive and Ubiquitous Computing, SECPerU 2007, Istanbul, Turkey, July 19, 2007*, pages 67–72. IEEE Computer Society, 2007.

52 Pascal Lafourcade, Denis Lugiez, and Ralf Treinen. Intruder deduction for AC-like equational theories with homomorphisms. In *Proc. 16th International Conference on Rewriting Techniques and Applications (RTA'05)*, volume 3467 of *LNCS*, pages 308–322. Springer, 2005.

53 G. Lowe. An attack on the Needham-Schroeder public key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1995.

54 Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.

55 R. Needham and M. Schroeder. Using encryption for authentification in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

56 Technical advisory group on machine readable travel documents (tag/mrtd). URL: `http://www.icao.int/Meetings/TAG-MRTD/TagMrtd22/TAG-MRTD-22_WP05.pdf`.

57 Sonia Santiago, Santiago Escobar, Catherine Meadows, and José Meseguer. A formal definition of protocol indistinguishability and its verification using Maude-NPA. In *Security and Trust Management*, pages 162–177. Springer, 2014.

58 Benedikt Schmidt, Simon Meier, C. J. F. Cremers, and David Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. In *Proc. 25th Computer Security Foundations Symposium (CSF'12)*, pages 78–94. IEEE Computer Society Press, 2012.

59 F. Javier Thayer, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(1):191–230, 1999.

60 Alwen Tiu. A trace based bisimulation for the spi calculus. In *Programming Languages and Systems*, pages 367–382. Springer, 2007.

61 Ton van Deursen and Sasa Radomirovic. Attacks on RFID protocols. *IACR Cryptology ePrint Archive - Report 2008/310*, 2008. URL: `http://eprint.iacr.org/2008/310`.

Formal Design, Implementation and Verification of Blockchain Languages

Grigore Rosu

University of Illinois at Urbana-Champaign and Runtime Verification, Inc., USA
`http://fsl.cs.illinois.edu/grosu`
grosu@illinois.edu

Abstract

This invited paper describes recent, ongoing and planned work on the use of the rewrite-based semantic framework $\mathbb{K}$ to formally design, implement and verify blockchain languages and virtual machines. Both academic and commercial endeavors are discussed, as well as thoughts and directions for future research and development.

1 Introduction and Motivation

Many of the recent expensive cryptocurrency bugs and exploits are due to flaws or weaknesses of the underlying blockchain programming languages or virtual machines [6, 4, 1, 3, 12]. The usual post-mortem approach to formal language semantics and verification, where the language is firstly implemented and used in production for many years before a need for formal semantics and verification tools naturally arises, does simply not work anymore. New blockchain languages or virtual machines are proposed at an alarming rate, followed by new versions of them every few weeks, sometimes every few days, together with programs (a.k.a. smart contracts) in these languages that are responsible for financial transactions totaling more than $1B/day only on the Ethereum blockchain [7]. Formal analysis and verification tools for such languages and virtual machines are therefore needed *immediately*.

In order to formally verify a program in any given language, a formal model of the program is necessary. Such a program model can be developed manually, in mechanical theorem provers such as Coq [10] or Isabelle [13], but this is usually expensive and thus

done rarely, mostly in the context of mission critical systems and in combination with other activities, such as defining model abstractions and protocol/algorithm/model validation. The norm is for tools to extract such program models automatically, based either on translations of the program to particular intermediate languages such as Boogie [2] or Why [8] that serve as input to specialized program verifiers, or on direct implementations of Hoare logics or verification condition (VC) generators for the target programming language.

The translation approach has the advantage that the same verifier can be used across various target languages, but it has the drawback that program behaviors may be lost in translation, so a trusted formal semantics of the original language and a proof of correctness of the translation are needed for increased confidence. Additionally, backwards translations of failed proofs need to be engineered, so users see error messages specific to their original language and not to the level of the intermediate language. The direct approach avoids both the translation correctness and the translation of failed proofs problems, but it is usually significantly more complex to implement, which can lead to subtle implementation errors that are hard or impossible to expose by testing (Hoare logics are not easily executable). Therefore, to increase confidence in such direct program verifiers, the underlying Hoare logic or VC generation procedure needs to be proved sound with respect to some trusted reference model of the target programming language, typically an operational semantics.

Therefore, a formal trusted semantics of the target programming language is required for increased confidence of program verification. The $\mathbb{K}$ framework [11] (`http://kframework.org`) takes the firm position that a formal language semantics should be needed to validate not only program verifiers, but essentially all the target language tools. Moreover and more importantly, that no other formal or informal, direct or indirect semantics of the target language should be required for any of the tools, and that the tools should be either generated automatically from or take as input the formal language semantics. That is, that all the language-specific tools for a given language should be produced automatically by the framework, correct-by-construction, from the formal semantics of the language. Figure 1 depicts the $\mathbb{K}$ belief and approach. This is nevertheless the best we can hope for in our field. But does it really work? Isn't it too idealistic? Aren't the tools too inefficient to be practical?

Some initial practical instances of the $\mathbb{K}$ approach were reported in [5], where existing formal semantics of C, Java and JavaScript were used as inputs to $\mathbb{K}$'s language-parametric program verifier, to yield program verifiers specific to these three languages. The resulting program verifiers were comparable in performance with existing state-of-the-art verifiers developed specifically for these languages. Here we bring additional evidence for the feasibility of the $\mathbb{K}$ approach, this time in the context of the blockchain. Specifically, we discuss recent academic and commercial results in designing blockchain languages and virtual machines by formalizing their semantics. Implementations for these are generated automatically from their semantics, in a correct-by-construction fashion, and so are program verifiers for them.

Why target the blockchain as an application domain for the $\mathbb{K}$ approach? First, because it is a new field in desperate need of formal verification; if cryptocurrencies are the future of money, then we ought to do our best to increase the security, safety and reliability of blockchain transactions. Second, because the entire blockchain space is a moving target, with paradigms and languages that change on a daily basis with no time to develop program verifiers following the traditional Hoare logic or VC generation approaches; therefore, it is a sweet spot for our language-parametric approach. Third, because two major blockchains holding cryptocurrencies, Ethereum and Cardano, showed unreserved interest in pushing formal methods in the design and implementation of their languages, and even deploy new versions of the blockchain using technology resulting from this research initiative. Finally,

```
module LAMBDA                                    syntax Exp ::= "if" Exp "then" Exp
  imports SUBSTITUTION                                          "else" Exp  [strict(1)]
  syntax Val ::= Id                              rule if true  then E else _ => E
               | "lambda" Id "." Exp  [binder]   rule if false then _ else E => E
  syntax Exp ::= Val
               | Exp Exp         [strict, left]  syntax Exp ::= "let" Id "=" Exp "in" Exp
               | "(" Exp ")"           [bracket] rule let X=E in E' => (lambda X . E') E
  syntax KVariable ::= Id
  syntax KResult ::= Val                         syntax Exp ::=
                                                      "letrec" Id Id "=" Exp "in" Exp
  rule (lambda X:Id . E:Exp) V:Val => E[V / X]      | "mu" Id "." Exp  [binder]
                                                 rule letrec F:Id X = E in E'
  syntax Val ::= Int | Bool                        => let F = mu F . lambda X . E in E'
  syntax Exp ::= Exp "*" Exp    [strict, left]   rule mu X . E => E[(mu X . E) / X]
               | Exp "/" Exp          [strict]
               > Exp "+" Exp    [strict, left]   syntax Exp ::= "callcc" Exp  [strict]
               > Exp "<=" Exp         [strict]   syntax Val ::= cc(K)
  rule I1 * I2 => I1 *Int I2                      rule <k> (callcc V:Val => V cc(K))
  rule I1 / I2 => I1 /Int I2                                -> K </k>
  rule I1 + I2 => I1 +Int I2                      rule <k> cc(K) V -> _  =>  V -> K </k>
  rule I1 <= I2 => I1 <=Int I2                   endmodule
```

■ **Figure 1** Left: the $\mathbb{K}$ framework approach to language design, implementation and verification. Center and right: the $\mathbb{K}$ definition of a call-by-value lambda calculus with arithmetic and callcc.

because it offers an environment where a language framework like $\mathbb{K}$ can be pushed even beyond its original, already ambitious goal: it can serve as a universal language of languages, where language semantics (or more exactly their hashes) are stored on the blockchain, and then correct-by-construction compilers and interpreters for such languages are generated automatically; this way, smart contract developers can program and verify them using their favorite languages, provided that they have a formal semantics on the blockchain.

2 $\mathbb{K}$ Framework

$\mathbb{K}$ is a rewrite-based executable semantic framework in which programming languages, type systems and formal analysis tools can be defined using *configurations*, *computations* and *rules*. Configurations organize the state in units called cells, which are labeled and can be nested. Computations carry computational meaning as special nested list structures sequentializing computational tasks, such as fragments of program. Computations extend the original language abstract syntax. K (rewrite) rules make it explicit which parts of the term they read-only, write-only, read-write, or do not care about. This makes K suitable for defining truly concurrent languages even in the presence of sharing. Computations are like any other terms in a rewriting environment: they can be matched, moved from one place to another, modified, or deleted. This makes K suitable for defining control-intensive features such as abrupt termination, exceptions or call/cc. Figure 1 left depicts the $\mathbb{K}$ architecture.

Figure 1 center and right shows the complete $\mathbb{K}$ definition of a simple call-by-value lambda calculus language with builtin arithmetic, conditional, let, letrec, and call/cc. Note that syntax is define using conventional BNF, with terminals in quotes. The | separates production of same precedence, while > states that the previous productions bind tighter than the subsequent ones. A parser is generated automatically and is used to parse both the programs and the semantic rules; i.e., rules can use concrete syntax. Syntax and rule declarations can be tagged with attributes. Some attributes have meaning for the parser, such as `left` for left associativity, others have semantic meaning, such as `binder` (used by the builtin variable-capture free substitution) and `strict` (which defines appropriate evaluation contexts). For $\mathbb{K}$'s internal substitution to work out of the box, we also need to tell it which syntactic categories act as variables, by subsorting them to `KVariable`. Similarly, for efficiency we need to tell it which categories build non-reducible results by subsorting to `KResult`. Most of the semantic rules are self-explanatory. The call/cc rules use $\mathbb{K}$'s specific *local rewriting*: rewriting takes place in context, specifically in the `<k/>` cell, and not at the top level. This gives $\mathbb{K}$ additional convenience and modularity in language definitions.

Taking such formal language definitions as input, $\mathbb{K}$ generates a variety of tools for the defined language as shown in Figure 1, without any other piece of knowledge about the given language except its formal syntax and semantics. Complete languages semantics for real-world languages like C, Java and JavaScript have been defined this way, and tools for them have been generated and shown to have acceptable performance when compared to existing adhoc tools for the same languages [5].

3   Current Progress

KEVM: Ethereum, the second largest blockchain cryptocurrency after Bitcoin, implements a general-purpose replicated "world computer" that allows for the development of arbitrary programs, called "smart contracts", that execute in blockchain transactions using the block-chain to synchronize their state globally. Smart contracts are written in various high-level languages, but are ultimately translated to a low-level language called the Ethereum Virtual Machine (EVM) [14]. Among other features, these contracts can tally user votes, communicate with other contracts, store or represent digital assets, and send or receive money in cryptocurrencies, without requiring trust in any third party to faithfully execute the contract. Their correct and secure operation relies entirely on the correctness of their EVM code. Any code error can be immediately exploited resulting in significant financial loss [6, 4, 1, 3, 12].

To enable the formal verification of smart contracts, in a project partially funded by the research and engineering company IOHK (`http://iohk.io`, the creators of the Cardano blockchain and the ADA cryptocurrency), we have formalized the semantics of the EVM [9]. Our $\mathbb{K}$ semantics of the EVM, which we refer to as KEVM, is as *complete* as it can be. We know this because we *tested* it by running the automatically generated interpreter (see Figure 1) against the comprehensive 40,000-program test suite that comes with the official C++ implementation of the EVM, which serves as a conformance suite for EVM implementations. Building upon KEVM, the startup Runtime Verification has formally verified several smart contracts as part of their commercial verification services (`https://runtimeverification.com/smartcontract/`).

A pleasant surprise was that the EVM interpreter automatically generated from KEVM turned out to be only *one order of magnitude* slower on average than the official C++ implementation offered by the Ethereum Foundation [9]. Since smart contracts are small and fast executing programs, the above suggests that KEVM can serve not only as a reference executable model of the EVM, but also as an actual production implementation. We are grateful to IOHK for launching a testnet on Cardano in Summer 2018 to test this hypothesis in a real-world setting. If successful, this experiment can be the first step towards a world where virtual machines are generated automatically from their formal specifications, correct-by-construction. If performance is not a problem, why should it be any other way?

IELE: One of the major lessons we learned during the EVM formalization effort was that EVM can be improved along various dimensions, improvements that could make both implementations and smart contract verification easier and faster. Instead of doing so, we preferred to design and implement a new virtual machine, IELE: `https://github.com/runtimeverification/iele-semantics`. Unlike the EVM, which is a stack-based machine, IELE is a register-based machine, like LLVM. IELE also directly supports functions, like LLVM, and is human readable. It has an unbounded number of registers and also supports unbounded integers. Like KEVM, the design of IELE was also done in a semantics-based style, using $\mathbb{K}$, and a VM was automatically generated from its formal specification. To our

knowledge, IELE is the first VM that was completely designed and implemented using formal methods. There is no line of low-level code written by humans; all code is automatically generated from its formal specification. The IELE project was funded by IOHK, with the explicit objective of eventually powering real-world blockchains, including Cardano. The project is complete and like with KEVM, a test net will be deployed in Summer 2018 by IOHK to evaluate IELE in a real-world setting. To make migration of existing Ethereum smart contracts to IELE possible, we have also developed a IELE compiler for the most commonly used smart contract language, Solidity: `https://github.com/runtimeverification/solidity`. We are currently also working on a IELE compiler for Plutus, a high-order functional programming language for future smart contracts developed by IOHK under the supervision of Philip Wadler: `https://github.com/kframework/plutus-core-semantics`.

Vyper, Casper: Vyper (`https://github.com/ethereum/vyper`) is a novel programming language for smart contracts that aims for increased security, simplicity, and human readability; Vyper currently compiles to the EVM. Casper (`https://github.com/ethereum/casper`) is a novel consensus protocol implemented in Vyper, meant to save wasteful electricity expenditures and at the same time provide greatly increased security. Vyper and Casper are both proposed by the Ethereum Foundation and, unfortunately, both are moving targets. Inconsistencies and bugs are found and fixed in Vyper on a weekly basis, which may potentially influence the Casper implementation, which itself still changes due to other forces. Funded by Ethereum Foundation, we have formalized the semantics of Vyper in $\mathbb{K}$, discovering several bugs and inconsistencies in the process: `https://github.com/kframework/vyper-semantics`; and we are also formally verifying the Casper code in Vyper, as compiled to EVM: `https://github.com/runtimeverification/verified-smart-contracts/tree/master/casper`. For verification purposes, we are currently regarding Casper as a smart contract eventually executed on the EVM, but its inherent complexity makes it highly non-trivial to correctly specify its intended behavior. To reduce the risk of misspecifying its Vyper code correctness, in a joint effort with the Etherum Foundation and the University of Texas at Austin we are also formalizing the actual protocol in Coq and in Isabelle, and validate the model by proving its intended safety and liveness properties. Then we will show that the proved Vyper code properties are consistent with the Coq and Isabelle models. All these are necessary due to the extremely important role that Casper will play in the near future for Ethereum.

4   Conclusion and Future Work

While $\mathbb{K}$ may not be the final answer to our quest for an ideal language framework, we believe that it has demonstrated that it is possible, and feasible, to generate a variety of formal execution and analysis tools for a given language from the formal semantics of that language. Moreover, only one, executable semantics for any given language suffices in order to generate all the tools, and that the so generated tools can be correct-by-construction, thus eliminating the need for redundant semantics and complex proofs of correctness.

Some years may still need to pass before sufficient evidence is accumulated to convince the skeptical formal methodist that the approach has merit even with mainstream languages like C and Java, for which well-engineered formal verification tools already exist. But for emerging fields like the blockchain, which come with new languages that routinely change every few days and require mostly small but tricky programs, the language-parametric semantic framework approach appears to be the only solution quickly available.

We hope that this wave of interest in language frameworks like 𝕂 will lead to the development of several important advances in the field, which will then be applicable across all languages. On the foundational side, we need to develop language-/paradigm-independent logics that allow us to specify any desired properties about any programs in any programming languages. On the practical side, automation is critical for the success of any verification environment. Also, generation of proof objects to act as correctness certificates for the various formal tools generated for a given language would increase demand and adoption of such tools, especially in the blockchain domain.

References

1 Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts. *IACR Cryptology ePrint Archive*, 2016:1007, 2016.

2 Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects (FMCO'05)*, volume 4111 of *LNCS*, pages 364–387, 2006.

3 Lorenz Breidenbach, Phil Daian, Ari Juels, and Emin Gün Sirer. An in-depth look at the parity multisig bug, 2017. URL: `http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/`.

4 Vitalik Buterin. Thinking about smart contract security, 2016. URL: `https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/`.

5 Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. Semantics-based program verifiers for all languages. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*. ACM, Nov 2016.

6 Phil Daian. DAO attack, 2016. URL: `http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/`.

7 Etherscan. Ethereum transactions, 2018. URL: `https://etherscan.io/`.

8 Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In *CAV*, volume 4590 of *LNCS*, pages 173–177, 2007.

9 Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, Brandon Moore, Yi Zhang, Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KEVM: A Complete Semantics of the Ethereum Virtual Machine. In *Computer Security Foundations Symposium (CSF'18)*, 2018. URL: `http://jellopaper.org`.

10 The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0. URL: `http://coq.inria.fr`.

11 Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

12 Jutta Steiner. Security is a process: A postmortem on the parity multi-sig library self-destruct, 2017. URL: `https://blog.ethcore.io/security-is-a-process-a-postmortem-on-the-parity-multi-sig-library-self-destruct/`.

13 The Isabelle development team. Isabelle, 2018. URL: `https://isabelle.in.tum.de/`.

14 Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014. (Updated for EIP-150 in 2017) `http://yellowpaper.io/`.

Challenges in Quantum Programming Languages

Peter Selinger

Dalhousie University, Halifax, Canada
selinger@mathstat.dal.ca
 https://orcid.org/0000-0003-3161-856X

Abstract

In this talk, I will give an overview of some recent progress and current challenges in the design of quantum programming languages. Unlike classical programs, which can in principle be debugged by stopping the program at critical moments and examining the contents of variables, quantum programs are not amenable to traditional debugging because the state of a quantum system cannot usually be examined in a meaningful way. Therefore, we need other methods for ensuring the correctness of quantum programs, such as formal verification. For this reason, I advocate the use of strongly typed, functional programming languages for quantum computing. As far as functional quantum programming languages are concerned, there is currently a relatively wide gap between theory and practice. On the one hand, we have languages with strong theoretical foundations, such as the quantum lambda calculus, which operate at a relatively low level of abstraction and lack many features that would be useful to practical quantum programmers. On the other hand, we have practical functional quantum programming languages such as Quipper, which is implemented as an embedded language in Haskell, has many high-level features, and has been used in large-scale projects, but lacks a theoretical basis and a strong type system [1, 2, 3, 6]. We have recently attempted to narrow this gap through a family of languages called Proto-Quipper, which are designed to offer Quipper-like features while having sound theoretical foundations [5, 4]. I will give an overview of Quipper and its most useful features, report on the progress we made with formalizing fragments of Quipper, and outline several of the still remaining challenges.

References

1 Alexander Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. An introduction to quantum programming in Quipper. In *Proceedings of the 5th International Conference on Reversible Computation, RC 2013, Victoria, British Columbia*, volume 7948 of *Lecture Notes in Computer Science*, pages 110–124. Springer, 2013. Also available from `arXiv:1304.5485`. `doi:10.1007/978-3-642-38986-3_10`.

2 Alexander Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: a scalable quantum programming language. In *Proceedings of the 34th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, Seattle*, volume 48(6) of *ACM SIGPLAN Notices*, pages 333–342, 2013. Also available from `arXiv:1304.3390`. `doi:10.1145/2499370.2462177`.

3 Alexander Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. The Quipper language. Software implementation, available from `http://www.mathstat.dal.ca/~selinger/quipper/`, 2013.

4   Francisco Rios and Peter Selinger. A categorical model for a quantum circuit description language. Extended abstract. In *Proceedings of the 14th International Workshop on Quantum Physics and Logic, QPL 2017, Nijmegen*, volume 266 of *Electronic Proceedings in Theoretical Computer Science*, pages 164–178. Open Publishing Association, 2018. Also available from `arXiv:1308.4557`. `doi:10.4204/EPTCS.266.11`.

5   Neil J. Ross. *Algebraic and Logical Methods in Quantum Computation*. PhD thesis, Department of Mathematics and Statistics, Dalhousie University, 2015. Available from `arXiv:1510.02198`.

6   Jonathan M. Smith, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: concrete resource estimation in quantum algorithms. Extended abstract for a talk given at the 12th International Workshop on Quantitative Aspects of Programming Languages and Systems, QAPL 2014, Grenoble. Available from `arXiv:1412.0625`, 2014.

Proof Techniques for Program Equivalence in Probabilistic Higher-Order Languages

Valeria Vignudelli
Univ Lyon, CNRS, ENS de Lyon, UCB Lyon 1, LIP
Lyon, France

Abstract

While the theory of functional higher-order languages, starting from lambda-calculi, is a well-established research field, it is only in recent years that the operational semantics of higher-order languages with probabilistic operators has started to be extensively studied. A fundamental notion in the semantics of programming languages is that of program equivalence. In higher-order languages, program equivalence is generally formalized as a contextual equivalence [6], which can be hard to prove directly. This has motivated the study of proof techniques for contextual equivalence, from inductive ones, such as logical relations [7], to coinductive ones, mainly in the form of bisimulations [1]. In this talk I will discuss proof techniques for program equivalence in languages combining higher-order and probabilistic features. Several operational methods, traditionally developed in a deterministic setting, have been adapted to probabilistic higher-order languages [2, 5, 3]. I will discuss these proof methods and focus on bisimulation-based techniques, showing how probabilities, combined with different language features, force a number of modifications to the definition of bisimulation [4, 8].

References

1 Samson Abramsky. The lazy lambda calculus. In David A. Turner, editor, *Research topics in functional programming*, pages 65–116. Addison-Wesley, 1990.

2 Ales Bizjak and Lars Birkedal. Step-indexed logical relations for probability. In *Proc. FoSSaCS'15*, pages 279–294, 2015.

3 Raphaëlle Crubillé and Ugo Dal Lago. On probabilistic applicative bisimulation and call-by-value λ-calculi. In *Proc. ESOP'14*, LNCS 8410, pages 209–228. Springer, 2014. `doi: 10.1007/978-3-642-54833-8_12`.

4 Raphaëlle Crubillé, Ugo Dal Lago, Davide Sangiorgi, and Valeria Vignudelli. On applicative similarity, sequentiality, and full abstraction. In Roland Meyer, André Platzer, and Heike Wehrheim, editors, *Correct System Design*, LNCS 9360, pages 65–82. Springer, 2015.

5 Ugo Dal Lago, Davide Sangiorgi, and Michele Alberti. On coinductive equivalences for higher-order probabilistic functional programs. In *Proc. POPL'14*, pages 297–308. ACM, 2014.

6   James H. Morris. *Lambda-calculus models of programming languages.* PhD thesis, Massachusetts Institute of Technology., 1968.

7   Andrew Pitts. Typed operational reasoning. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 7, pages 245–289. MIT Press, 2005.

8   Davide Sangiorgi and Valeria Vignudelli. Environmental bisimulations for probabilistic higher-order languages. In *Proc. POPL'16*, pages 595–607. ACM, 2016.

A Unifying Framework for Type Inhabitation

Sandra Alves[1]

DCC-Faculty of Science & CRACS, University of Porto, Portugal
sandra@dcc.fc.up.pt
🆔 https://orcid.org/0000-0001-8840-5587

Sabine Broda[2]

DCC-Faculty of Science & CMUP, University of Porto, Portugal
sbb@dcc.fc.up.pt
🆔 https://orcid.org/0000-0002-3798-9348

──── **Abstract** ────

In this paper we define a framework to address different kinds of problems related to type inhabitation, such as type checking, the emptiness problem, generation of inhabitants and counting, in a uniform way. Our framework uses an alternative representation for types, called the pre-grammar of the type, on which different methods for these problems are based. Furthermore, we define a scheme for a decision algorithm that, for particular instantiations of the parameters, can be used to show different inhabitation related problems to be in PSPACE.

1 Introduction

Inhabitation of simply typed λ-terms and its related problems, such as type checking, the emptiness problem, generation of inhabitants, counting algorithms, etc. have been extensively studied throughout the years [2, 3, 5, 15, 14, 10, 12, 13, 6], using a variety of formalisms such as context-free grammars [15], tree-based methods [4], automata theory [13], amongst others. Despite the diversity of methods, there are common fundamental features that emerge from the different approaches.

One of these features is the implicit relation between the structure of a type and its normal inhabitants. The *Formula-Tree Method* by Broda and Damas [4] explores this relation by looking at a tree representation of the type, identifying what are called the primitive parts, which are then combined following a set of rules determined by the structure of the type. In the case of the inhabitation machines defined by Schubert et al. [13], the states of the automata used to recognize the inhabitants of a given type, as well as the transition relation between configurations of the machines, are obtained directly from the sub-expressions of the type. More recently, while studying the complexity of the principal inhabitation problem, Dudenhefner and Rehof [6] use the structure of the type to define a path relation identifying subformulas with the same atomic type. This path relation is then used in the definition

of an algorithm that addresses principal inhabitation. Types and their structure are also fundamental in the definition of the context-free grammars by Takahashi et al. [15].

In this paper we highlight the importance of the underlying structure of types in the definition of methods for type inhabitation related problems. To that end, we present an alternative unifying representation of the type's structure, which we call the *pre-grammar* of the type. From this simple, yet powerful, device we extract rewriting methods to deal with type-checking, counting and generation of inhabitants.

Secondly, we explore the uniformity of decision algorithms defined over the years to prove that different inhabitation related problems are in PSPACE. Complexity of inhabitation related problems was first addressed by Statman [14] in the realm of propositional intuitionistic logic. The decidability of the logic was proved to be PSPACE complete, and therefore also the emptiness problem for the simply typed lambda calculus, due to the well-known Curry-Howard correspondence [11]. A direct syntactic proof of the same result, for the simply typed λ-calculus, was later given by Urzyczyn [16]. PSPACE completeness of the infiniteness problem was proved by Hirokawa [10], by reducing the emptiness problem to the infiniteness problem. In [6], PSPACE completeness was proved for the problem of principal inhabitation, by means of a non-deterministic algorithm for choosing a particular path relation for a given type. Also in the case of inhabitation machines [13], a PSPACE completeness result is obtained for the emptiness problem by means of a polynomial time alternating algorithm. In fact, several of the results mentioned above rely on polynomial time alternating algorithms. Note that the class of problems decidable in alternating polynomial time (AP) corresponds to the class of problems decidable in polynomial space (PSPACE). Following that, we define a scheme for a polynomial time alternating decision algorithm, which operates on the rules of the pre-grammar of the type. By instantiating the parameters of the algorithm scheme, we obtain different PSPACE decision algorithms for the problems of emptiness, counting and principal inhabitation.

We will restrict our methods and definitions to terms in normal form. In fact, most of the interesting questions related to inhabitation can be reduced to, or even just make sense for, normal terms. For instance, an inhabited type may have only a finite number of normal inhabitants, but has always an infinite number of (not necessarily normal) inhabitants. Also, every inhabited type is the principal type of an infinite number of terms, while it may not be the principal type of a term in normal form [9].

The rest of the paper is structured as follows. In the next section we introduce some preliminary notions. In Section 3, we present the notion of pre-grammars and prove some basic results. Using the pre-grammar representation, in Section 4, we define rewriting methods to address type checking and the emptiness problem, and explore closure properties, for intersection and union types. In Section 5 we define the scheme of an alternating decision algorithm, and its instances. Finally, in Section 6, we draw some conclusions and highlight some future work.

2 Preliminaries

In this paper we assume familiarity with the simply typed λ-calculus à la Curry [8]. We denote type variables (atoms) by $a, b, c, \ldots$ and arbitrary types by lower-case Greek letters $\alpha, \beta, \gamma, \sigma, \tau, \ldots$. The set of simple types is denoted by $\mathcal{T}$. We denote λ-terms by $M, N, \ldots$, which are built from an infinite countable set of term variables $\mathcal{V}$. Unless stated otherwise, we identify terms modulo α-equivalence. For type assignment we consider the system TA_λ as described in [8] and consider its inference rules for terms in β-normal form. Note that every

β-normal λ-term is of the form $\lambda x.N$ or $xN_1 \cdots N_s$, where $N, N_1, \ldots, N_s$ are in normal form and $s \geq 0$. Different from [8] we define the depth of a λ-term by $\mathsf{depth}(\lambda x.N) = 1 + \mathsf{depth}(N)$, and $\mathsf{depth}(xN_1 \cdots N_s) = 1 + max(\mathsf{depth}(N_1), \ldots, \mathsf{depth}(N_s))$ for $s \geq 1$, and $\mathsf{depth}(x) = 1$. With this definition the depth of a term M, such that $\Gamma \vdash M : \tau$, corresponds directly to the height of the unique TA_λ-deduction of this fact, as defined below. A *context* is a finite set Γ of declarations $x : \sigma$, where $x \in \mathcal{V}$ and $\sigma \in \mathcal{T}$, such that all term variables occurring in Γ are distinct from each other. The set of term variables occurring in Γ is denoted by $\mathsf{Subj}(\Gamma)$. The union of contexts is *consistent* if it does not contain different type declarations for the same term variable.

▶ **Definition 1.** We write $\Gamma \vdash M : \tau$ and say that type τ can be assigned to term M in context Γ, if this formula can be obtained by applying the rules below a finite number of times.
- If $\Gamma \vdash N : \sigma_2$ and $\Gamma \cup \{x : \sigma_1\}$ is consistent, then $\Gamma \setminus \{x : \sigma_1\} \vdash \lambda x.N : \sigma_1 \to \sigma_2$.
- if $\Gamma_i \vdash N_i : \sigma_i$, for $1 \leq i \leq s$ ($s \geq 0$), then $\Gamma \vdash xN_1 \cdots N_s : \sigma$, if $\Gamma = \Gamma_1 \cup \cdots \cup \Gamma_s \cup \{x : \sigma_1 \to \cdots \to \sigma_s \to \sigma\}$ is consistent;

If $\Gamma = \emptyset$, then we also write $\vdash M : \tau$ instead of $\Gamma \vdash M : \tau$ and say that M is an *inhabitant* of type τ. The set of all (normal) inhabitants of τ is denoted by $\mathsf{Nhabs}(\tau)$.

One knows that $\Gamma \vdash M : \tau$ implies that the set of term variables in Γ coincides with the set of free variables in M, i.e. $\mathsf{Subj}(\Gamma) = \mathsf{FV}(M)$, cf. Lemma 2A10 in [8]. Furthermore, for every derivable formula $\Gamma \vdash M : \tau$ there is exactly one deduction in TA_λ.

▶ **Example 2.** Consider type $\alpha = ((o \to o) \to o \to o) \to o \to o$, which will be our running example throughout this paper. Normal inhabitants of α are, for instance, $M_1 = \lambda xy.x(\lambda z.y)y$ and $M_2 = \lambda x.x(\lambda y.y)$, for which one has $\mathsf{depth}(M_1) = 5$ and $\mathsf{depth}(M_2) = 4$.

▶ **Definition 3.** The polarity of occurrences of subtypes in a type τ is defined as follows.
- τ is a positive occurrence in τ;
- if $\rho \to \sigma$ occurs positively (resp. negatively) in τ, then that occurrence of ρ is negative (resp. positive) and that occurrence of σ is positive (resp. negative) in τ.

Following the notation in [8], we will on occasions write $\underline{o}$ when referring to a particular occurrence of an object o. Every type τ can be uniquely written as $\tau = \tau_1 \to \ldots \to \tau_l \to a$, where a is a type variable and $l \geq 0$. Type variable a is called the *tail* of τ and denoted by $\mathsf{tail}(\tau)$. If $l \geq 1$, then $\tau_1, \ldots, \tau_l$ are called the *arguments* of τ. An occurrence $\underline{\sigma}$ in τ is called a *negative subpremise* of τ iff it is the argument of a positive occurrence of a subtype in τ.

Consider a term M and a type τ such that $\vdash M : \tau$, as well as a formula $\Gamma \vdash N : \sigma$, appearing in the unique TA_λ-deduction of $\vdash M : \tau$. In the following, we assign to each $X \in \mathsf{Subj}(\Gamma) \cup \{N\}$ an occurrence $\mathsf{st}(X)$ of a subtype in τ. The definition of st is bottom-up, starting with $\vdash M : \tau$.
- For $\vdash M : \tau$, let $\mathsf{st}(M) = \tau$.
- Now consider $\Gamma \setminus \{x : \sigma_1\} \vdash \lambda x.N : \sigma_1 \to \sigma_2$, because $\Gamma \vdash N : \sigma_2$ and because $\Gamma \cup \{x : \sigma_1\}$ is consistent. Consider $\mathsf{st}(\lambda x.N) = \underline{\sigma_1 \to \sigma_2}$ for $\Gamma \setminus \{x : \sigma_1\} \vdash \lambda x.N : \sigma_1 \to \sigma_2$. Then, for $\Gamma \vdash N : \sigma_2$ let $\mathsf{st}(N)$ be the occurrence of σ_2 in $\mathsf{st}(\lambda x.N)$. If $x \in \mathsf{Subj}(\Gamma)$, then $\mathsf{st}(x)$ is the occurrence of σ_1 in $\underline{\sigma_1 \to \sigma_2}$. All other variables in $\mathsf{Subj}(\Gamma)$ are assigned the same occurrences as for the formula $\Gamma \setminus \{x : \sigma_1\} \vdash \lambda x.N : \sigma_1 \to \sigma_2$.
- Finally let $\Gamma \vdash xN_1 \cdots N_s : \sigma$, because $\Gamma_i \vdash N_i : \sigma_i$, for $1 \leq i \leq s$ ($s \geq 0$), and because $\Gamma = \Gamma_1 \cup \cdots \Gamma_s \cup \{x : \sigma_1 \to \cdots \to \sigma_s \to \sigma\}$ is consistent. If $\mathsf{st}(x) = \underline{\sigma_1 \to \cdots \to \sigma_s \to \sigma}$, then $\mathsf{st}(N_i)$ is the occurrence of σ_i in $\mathsf{st}(x)$, for $\Gamma_i \vdash N_i : \sigma_i$ and $1 \leq i \leq s$ ($s \geq 0$). The variables in $\mathsf{Subj}(\Gamma_i)$ are assigned the same occurrences as for $\Gamma \vdash xN_1 \cdots N_s : \sigma$.

The following lemma, cf. [4], establishes the relationship between occurrences of variables in abstraction sequences and occurrences of subterms in M, respectively with negative subpremises and positive occurrences of subtypes in τ and can be easily proved using the definition of st above, as well as Definition 3. The established relationship will be explored in the definition of pre-grammars in the next section.

▶ **Lemma 4.** *Consider a term M in β-normal form and a type τ such that $\vdash M : \tau$, as well as a formula $\Gamma \vdash N : \sigma$, appearing in the unique TA_λ-deduction of $\vdash M : \tau$. If $x : \sigma_x \in \Gamma$, then $\mathsf{st}(x) = \underline{\sigma_x}$ is a negative subpremise in τ. Furthermore, $\mathsf{st}(N) = \underline{\sigma}$ is a positive occurrence of subtype σ in τ.*

3   Pre-grammars

In this section we describe how to obtain for a type τ a set of rewriting rules, which we call the *pre-grammar* of τ and denote by $\mathsf{pre}(\tau)$. We start by associating to each type τ a set $\mathsf{occT}(\tau)$ that contains for each type occurrence $\underline{\sigma}$ a tuple (σ, n, l), where $n \in \mathbb{N}$, and $\mathsf{l} \in \{\mathsf{var}\} \cup \{\, n \to m \mid n, m \in \mathbb{N} \,\}$. Distinct occurrences of subtypes are assigned distinct tuples. This set is uniquely defined, up to isomorphism between integers used in the tuples.

▶ **Definition 5.** Given a type $\tau \in \mathcal{T}$ let $\mathsf{occT}(\tau)$ be the smallest set satisfying the following.
- For each occurrence of a type variable a in τ there is a tuple $(a, n, \mathsf{var}) \in \mathsf{occT}(\tau)$;
- if $\rho \to \sigma$ is an occurrence of a subtype of τ, and $(\rho, n, \mathsf{l}_\rho), (\sigma, m, \mathsf{l}_\sigma) \in \mathsf{occT}(\tau)$ are the tuples corresponding to ρ and σ in this occurrence, then $(\rho \to \sigma, k, n \to m) \in \mathsf{occT}(\tau)$;
- for each $n \in \mathbb{N}$ there is at most one tuple $(\sigma, n, \mathsf{l}) \in \mathsf{occT}(\tau)$.

Furthermore, given a particular occurrence $\underline{\sigma}$ of a subtype of τ we denote by $\mathsf{n}(\underline{\sigma})$ the unique integer n such that $(\sigma, n, \mathsf{l}) \in \mathsf{occT}(\tau)$. We frequently will refer to $\mathsf{n}(\sigma)$ as the *identifier* of $\underline{\sigma}$ w.r.t. $\mathsf{occT}(\tau)$. Finally, $\mathsf{t}(n) = \sigma$, $\mathsf{lab}(n) = \mathsf{l}$, and $\mathsf{N}(\tau) = \{\, n \mid (\sigma, n, \mathsf{l}) \in \mathsf{occT}(\tau) \,\}$.

In order to deal correctly with the correspondence between occurrences of subtypes and occurrences of subterms, polarities have to be taken into account. With this purpose, and whenever convenient, we might superscript an integer n with '+' if n corresponds to a positive occurrence of a subtype, i.e. an occurrence that can be the type of a subterm of an inhabitant, and with '−' if it corresponds to a negative subpremise, i.e. if it corresponds to an occurrence that can be the type of a variable in an abstraction sequence. Integers that correspond to a negative occurrence, which is no subpremise, will not be superscripted.

▶ **Definition 6.** We say that two integers $n, m \in \mathsf{N}(\tau)$ are equivalent w.r.t. $\mathsf{occT}(\tau)$, and write $n \equiv_{\mathsf{occT}} m$, if and only if $\mathsf{t}(n) = \mathsf{t}(m)$. The binary relation $T(\tau) \subseteq \mathsf{N}(\tau) \times \mathsf{N}(\tau)$ is defined by $(p_2, p_3) \in T(\tau)$ iff $(\beta, p_3, p_1 \to p_2) \in \mathsf{occT}(\tau)$, i.e. $\beta = \beta_1 \to \beta_2$, $\mathsf{n}(\beta_1) = p_1$, $\mathsf{n}(\beta_2) = p_2$, and $\mathsf{n}(\beta) = p_3$. Furthermore, for $(p_2, p_3) \in T(\tau)$ let $\mathsf{q}(p_2, p_3) = p_1$.

▶ **Lemma 7.** *If τ contains s occurrences $\underline{a_1}, \ldots, \underline{a_s}$, of type variables, then the graph of $T(\tau)$, whose set of nodes is $\mathsf{N}(\tau)$, consists of s unary trees with roots $\mathsf{n}(a_1), \ldots, \mathsf{n}(a_s)$, respectively.*

▶ **Example 8.** For $\alpha = ((o \to o) \to o \to o) \to o \to o$ from Example 2 the set $\mathsf{occT}(\alpha)$ contains eleven tuples (β, n, l), where β, n and l are given below.

β	n	l
o	0	var
o	1	var
o	2	var
o	3	var

β	n	l
o	4	var
o	5	var
$o \to o$	6	$0 \to 1$
$o \to o$	7	$2 \to 3$

β	n	l
$o \to o$	8	$4 \to 5$
$(o \to o) \to o \to o$	9	$6 \to 7$
α	10	$9 \to 8$

The equivalence relation $\equiv_{\mathsf{occT}}$ partitions $\mathsf{N}(\alpha)$ into four equivalence classes, which are $\{10^+\}$, $\{9^-\}$, $\{6^+, 7, 8^+\}$, and $\{0^-, 1^+, 2^+, 3, 4^-, 5^+\}$. The associated graph $T(\alpha)$ is depicted below.

$$
\begin{array}{cccccc}
10^+ & & & 9^- & & \\
9\uparrow & & & 6\uparrow & & \\
8^+ & 6^+ & & 7 & & \\
4\uparrow & 0\uparrow & & 2\uparrow & & \\
5^+ & 1^+ & 2^+ & 3 & 4^- & 0^-
\end{array}
$$
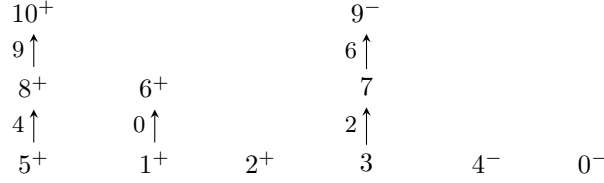
Now, $\mathsf{pre}(\tau)$ can be computed from $\mathsf{occT}(\tau)$ and $T(\tau)$ as follows.

▶ **Definition 9.** Given a type τ and a set of tuples $\mathsf{occT}(\tau)$, we denote by $\mathsf{pre}(\tau)$ the smallest set of rules satisfying the following conditions.

- If $m^+, k^-, n^+ \in \mathsf{N}(\tau)$, $(\beta, m, k \to n) \in \mathsf{occT}(\tau)$, then $m := \lambda k.n \in \mathsf{pre}(\tau)$;
- if $m^+, p_0^- \in \mathsf{N}(\tau)$ and $(p_s, p_{s-1}), \ldots, (p_2, p_1), (p_1, p_0) \in T(\tau)$, for some $s \geq 0$, $m^+ \equiv_{\mathsf{occT}} p_s$, $\mathsf{q}(p_i, p_{i-1}) = n_i$ for $1 \leq i \leq s$, then $m := p_0 \, n_1 \cdots n_s \in \mathsf{pre}(\tau)$.

▶ **Note 10.** If τ is inhabited, then there is exactly one rule for $\mathsf{n}(\tau)$ in $\mathsf{pre}(\tau)$. This rule is of the form $\mathsf{n}(\tau) := \lambda k.n$, for some $k^-, n^+ \in \mathsf{N}(\tau)$. Also, $\mathsf{n}(\tau)$ occurs in no other rule.

It is straightforward to verify the following two properties of $\mathsf{pre}(\tau)$.

▶ **Lemma 11.**
1. *Consider a positive occurrence of a subformula $\rho \to \sigma$ in τ and the corresponding tuple in $(\rho \to \sigma, m, k \to n) \in \mathsf{occT}(\tau)$. Then, $m := \lambda k.n \in \mathsf{pre}(\tau)$, and there is no other rule of the form $m := \lambda k'.n'$ in $\mathsf{pre}(\tau)$.*
2. *Consider a negative subpremise $\rho = \sigma_1 \to \cdots \to \sigma_s \to \sigma$ in τ and let $(\sigma_1, n_1, \mathsf{l}_1), \ldots, (\sigma_s, n_s, \mathsf{l}_s), (\sigma, n, \mathsf{l}_n), (\rho, k, \mathsf{l}_k)$ be the tuples in $\mathsf{occT}(\tau)$ corresponding to $\sigma_1, \ldots, \sigma_s, \sigma, \rho$, respectively. If $m^+ \in \mathsf{occT}(\tau)$, such that $m^+ \equiv_{\mathsf{occT}} n$, then $m := k \, n_1 \cdots n_s \in \mathsf{pre}(\tau)$. Furthermore, there is no other rule of the form $m := k \, n_1' \cdots n_t'$ $(t \geq 0)$ in $\mathsf{pre}(\tau)$.*

▶ **Example 12.** From $\mathsf{occT}(\alpha)$ and $T(\alpha)$ in Example 8 we obtain the following set $\mathsf{pre}(\alpha)$ containing fourteen rewriting rules.

10	:=	$\lambda 9.8$	6	:=	$\lambda 0.1 \mid 9\,6$	2	:=	$9\,6\,2 \mid 4 \mid 0$
8	:=	$\lambda 4.5 \mid 9\,6$	5	:=	$9\,6\,2 \mid 4 \mid 0$	1	:=	$9\,6\,2 \mid 4 \mid 0$

4 Inhabitation

4.1 Type Checking

In the following we describe a rewriting algorithm that, given a type τ and a term M, verifies if $\vdash M : \tau$, i.e. checks if $M \in \mathsf{Nhabs}(\tau)$. During the rewriting process we use objects with the structure of λ-terms, but such that integers can be used as placeholders for variables. We refer to these objects as *extended terms*. We denote by $N[k/x]$ the (extended) term obtained from N by replacing all free occurrences of variable x in N by placeholder k.

▶ **Definition 13.** Given a type τ, we write $(M, m) \hookrightarrow (N_1, n_1) \cdots (N_s, n_s)$, $(s \geq 0)$, where $M, N_1, \ldots, N_s$ are extended terms and $m, n_1, \ldots, n_s \in \mathsf{N}(\tau)$, if one of the following applies.

- If $m := \lambda k.n \in \mathsf{pre}(\tau)$, then $(\lambda x.N, m) \hookrightarrow (N[k/x], n)$;
- if $m := k \, n_1 \cdots n_s \in \mathsf{pre}(\tau)$, then $(k \, N_1 \cdots N_s, m) \hookrightarrow (N_1, n_1), \ldots, (N_s, n_s)$.

The definition of $\hookrightarrow$ extends, in the usual way, to rewriting of sequences of pairs, where we assume that sequences of pairs are processed from left to right. Then, $\hookrightarrow^*$ denotes the reflexive, transitive closure of $\hookrightarrow$.

Note that, by Lemma 11, in each step of $(M, \mathsf{n}(\tau)) \hookrightarrow^* \epsilon$, at most one rule of $\mathsf{pre}(\tau)$ applies to each pair. Consequently, the type-checking algorithm is deterministic and the sequence from $(M, \mathsf{n}(\tau))$ to ϵ is unique.

▶ **Example 14.** Consider α as before and $M_1 = \lambda xy.x(\lambda z.y)y$ from Example 2. Then,

$$
\begin{aligned}
(\lambda xy.x(\lambda z.y)y, 10) \quad &\hookrightarrow \quad (\lambda y.9(\lambda z.y)y, 8) \hookrightarrow (9(\lambda z.4)4, 5) \\
&\hookrightarrow \quad (\lambda z.4, 6), (4, 2) \hookrightarrow (4, 1), (4, 2) \hookrightarrow (4, 2) \hookrightarrow \epsilon.
\end{aligned}
$$

▶ **Theorem 15.** $\mathsf{Nhabs}(\tau) = \{\, M \mid (M, \mathsf{n}(\tau)) \hookrightarrow^* \epsilon \,\}$.

Proof. We show that for any term M, context $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ and type σ, if $\Gamma \vdash M : \sigma$, then $(M[\mathsf{n}(\sigma_1)/x_1, \cdots, \mathsf{n}(\sigma_n)/x_n], \mathsf{n}(\sigma)) \hookrightarrow^* \epsilon$, using $M[\Gamma]$ as an abbreviation for $M[\mathsf{n}(\sigma_1)/x_1, \cdots, \mathsf{n}(\sigma_n)/x_n]$. As a consequence it follows that $\mathsf{Nhabs}(\tau) \subseteq \{\, M \mid (M, \mathsf{n}(\tau)) \hookrightarrow^* \epsilon \,\}$. We proceed by induction on $\mathsf{depth}(M)$. First, consider $M = xN_1 \cdots N_s$ and suppose that $\Gamma \vdash xN_1 \cdots N_s : \sigma$, because $\Gamma_i \vdash N_i : \sigma_i$, for $1 \le i \le s$ ($s \ge 0$), and because $\Gamma = \Gamma_1 \cup \cdots \cup \Gamma_s \cup \{x : \sigma_1 \to \cdots \to \sigma_s \to \sigma\}$ is consistent. By Lemma 4, we know that there is a negative subpremise $\mathsf{st}(x) = \underline{\sigma_1 \to \cdots \to \sigma_s \to \sigma}$ in τ, as well as a positive occurrence $\mathsf{st}(xN_1 \cdots N_s)$ of σ in τ, corresponding to $\Gamma \vdash xN_1 \cdots N_s : \sigma$. Let n and m be respectively the identifier of the occurrence of σ in $\mathsf{st}(x)$, and of the positive occurrence $\mathsf{st}(xN_1 \cdots N_s)$ of σ in τ. Then, $m^+ \equiv_{\mathsf{occT}} n$ and it follows from Lemma 11 that $m := k\, \mathsf{n}(\sigma_1) \cdots \mathsf{n}(\sigma_s) \in \mathsf{pre}(\tau)$, where $k = \mathsf{n}(\sigma_1 \to \cdots \to \sigma_s \to \sigma)$. Thus $(M[\Gamma], m) \hookrightarrow (N_1[\Gamma], \mathsf{n}(\sigma_1)), \ldots, (N_s[\Gamma], \mathsf{n}(\sigma_s))$. But $N_i[\Gamma] = N_i[\Gamma_i]$, for $1 \le i \le s$. Consequently, the result follows from the induction hypothesis. Now, consider $M = \lambda x.N$ and suppose that we have $\Gamma \setminus \{x : \sigma_1\} \vdash \lambda x.N : \sigma_1 \to \sigma_2$, because $\Gamma \vdash N : \sigma_2$ and $\Gamma \cup \{x : \sigma_1\}$ is consistent. It follows from Lemma 4 that $\mathsf{st}(\lambda x.N) = \underline{\sigma_1 \to \sigma_2}$ is a positive occurrence of $\sigma_1 \to \sigma_2$ in τ. We consider the corresponding tuple $(\sigma_1 \to \sigma_2, m, k \to n) \in \mathsf{occT}(\tau)$, where $\mathsf{n}(\sigma_1 \to \sigma_2) = m$, $\mathsf{n}(\sigma_1) = k$, and $\mathsf{n}(\sigma_2) = n$. By Lemma 11, there is a rule $m := \lambda k.n \in \mathsf{pre}(\tau)$. Furthermore, $\mathsf{Subj}(\Gamma) = \mathsf{FV}(N)$ and $\mathsf{Subj}(\Gamma \setminus \{x : \sigma_1\}) = \mathsf{FV}(\lambda x.N)$. Thus, we have $(M[\Gamma \setminus \{x : \sigma_1\}], m) \hookrightarrow (N[\Gamma], n)$. The result follows from the induction hypothesis.

For the other inclusion consider a term M, such that $(M, \mathsf{n}(\tau)) \hookrightarrow^* \epsilon$. Let (E, p) be any pair appearing in the corresponding rewriting sequence, where E is an extended term and $p = \mathsf{n}(\sigma)$, for some type occurrence $\underline{\sigma}$ in τ, i.e. $\sigma = \mathsf{t}(p)$. Naturally, we have $(E, p) \hookrightarrow^* \epsilon$. Let $P = \{p_1, \ldots, p_l\}$ be the set of placeholders that occur in E. Furthermore, let us interpret each of the integers in P as the name of a term variable. We will show, by induction on the length of $(E, p) \hookrightarrow^* \epsilon$, that $\Gamma_P \vdash E : \mathsf{t}(p)$, where $\Gamma_P = \{p_1 : \mathsf{t}(p_1), \ldots, p_l : \mathsf{t}(p_l)\}$. In particular, it follows that $\vdash M : \tau$. First, consider $E = \lambda x.E'$ such that $(\lambda x.E', p) \hookrightarrow (E'[k/x], n)$ by rule $p := \lambda k.n \in \mathsf{pre}(\tau)$. This means that there is a positive occurrence of a subtype $\mathsf{t}(p) = \mathsf{t}(k) \to \mathsf{t}(n)$ in τ. By the induction hypothesis, we have $\Gamma_{E'} \cup \{k : \mathsf{t}(k)\} \vdash E'[k/x] : \mathsf{t}(n)$. Thus, $\Gamma_{E'} \vdash \lambda k.E'[k/x] : \mathsf{t}(p)$, but $\lambda k.E'[k/x] \equiv_\alpha \lambda x.E'$. Finally, consider $E = k\, E_1 \cdots E_s$, such that $(k\, E_1 \cdots E_s, p) \hookrightarrow (E_1, n_1), \ldots, (E_s, n_s)$ by rule $p := k\, n_1 \cdots n_s \in \mathsf{pre}(\tau)$, where $s \ge 0$. Then, $\mathsf{t}(k) = \mathsf{t}(n_1) \to \cdots \to \mathsf{t}(n_s) \to \mathsf{t}(p)$ is a negative subpremise in τ. By the induction hypothesis, we have $\Gamma_{E_i} \vdash E_i : \mathsf{t}(n_i)$, for $1 \le i \le s$ and $s \ge 0$. Furthermore, $\Gamma_E = \Gamma_{E_1} \cup \cdots \cup \Gamma_{E_s} \cup \{k : \mathsf{t}(k)\}$ is consistent by definition. Thus, $\Gamma_E \vdash E : \mathsf{t}(p)$. ◀

4.2 The Emptiness Problem

In this subsection we define a rewriting algorithm to decide if a given type τ has a normal inhabitant. Contrary to the previous one, this algorithm is non-deterministic, since more than one rule may apply at each step. On the other hand, it provides us with a simple tool to show that the emptiness problem for simple types is in PSPACE, and can be used for generation as well as for counting.

▶ **Definition 16.** Given a type τ, an identifier $m \in \mathsf{N}(\tau)$ and a set $V \subseteq \mathsf{N}(\tau)$, we write $(m, V) \rightsquigarrow (n_1, V'), \ldots, (n_s, V')$ if one of the following applies.

▬ If $m := \lambda k.n \in \mathsf{pre}(\tau)$, then $(m, V) \rightsquigarrow (n, V \cup \{k\})$;

▬ if $m := k\ n_1 \cdots n_s \in \mathsf{pre}(\tau)$ and $k \in V$, then $(m, V) \rightsquigarrow (n_1, V), \ldots, (n_s, V)$.

The definition of $\rightsquigarrow$ extends, in the usual way, to rewriting of sequences of pairs. Then, $\rightsquigarrow^*$ denotes the reflexive, transitive closure of $\rightsquigarrow$.

▶ **Definition 17.** For a particular rewriting sequence of $(\mathsf{n}(\tau), \emptyset) \rightsquigarrow^* \epsilon$, we define a function pair that computes for each (m, V) in that rewriting sequence a tuple $(M, \Gamma) = \mathsf{pair}(m, V)$. For convenience we will use identifiers as indexes of term variables in such a way that the type assigned to a variable with name x_n, for $n \in \mathsf{N}(\tau)$, is always $\mathsf{t}(n)$. The function pair is recursively defined as follows.

▬ If $(m, V) \rightsquigarrow (n, V \cup \{k\})$ because $m := \lambda k.n \in \mathsf{pre}(\tau)$, then $\mathsf{pair}(m, V) = (\lambda x_k.N, \Gamma \setminus \{x_k : \mathsf{t}(k)\})$, where $(N, \Gamma) = \mathsf{pair}(n, V \cup \{k\})$;

▬ if $(m, V) \rightsquigarrow (n_1, V), \ldots, (n_s, V)$ because $m := k\ n_1 \cdots n_s \in \mathsf{pre}(\tau)$ and $k \in V$, then $\mathsf{pair}(m, V) = (x_k\ N_1 \cdots N_s, \{x_k : \mathsf{t}(k)\} \cup \Gamma_1 \cup \cdots \cup \Gamma_s)$, where $(N_i, \Gamma_i) = \mathsf{pair}(n_i, V)$, for $1 \leq i \leq s\ (s \geq 0)$.

Note that function pair actually does not depend on set V, but on the identifier m and on the rule in $\mathsf{pre}(\tau)$, which is used in each step of the rewriting sequence. The rule is implicitly given by the pairs appearing on the right of $\rightsquigarrow$ in Definition 16, unless it is of the form $m := k$. In that case $(m, V) \rightsquigarrow \epsilon$ and there might be more than one identifier $k \in V$ such that $m := k \in \mathsf{pre}(\tau)$. To guarantee that pair is well-defined, we suppose that in each rewriting step, the corresponding rewriting rule is given, either implicitly or explicitly. The correctness of function pair is stated in the following lemma.

▶ **Lemma 18.** *If $(m, V) \rightsquigarrow^* \epsilon$ and $(M, \Gamma) = \mathsf{pair}(m, V)$ for some corresponding rewriting sequence, then $\Gamma \vdash M : \mathsf{t}(m)$.*

Proof. By structural induction on M. We first consider the case where $\mathsf{pair}(m, V) = (\lambda x_k.N, \Gamma \setminus \{x_k : \mathsf{t}(k)\})$, which follows from $(m, V) \rightsquigarrow (n, V \cup \{k\}) \rightsquigarrow^* \epsilon$ because $m := \lambda k.n \in \mathsf{pre}(\tau)$ and $(N, \Gamma) = pair(n, V \cup \{k\})$. By the induction hypothesis, $\Gamma \vdash N : \mathsf{t}(n)$ and by definition $\Gamma \cup \{x_k : \mathsf{t}(k)\}$ is always consistent. Therefore, $\Gamma \vdash \lambda x_k.N : \mathsf{t}(k) \rightarrow \mathsf{t}(n)$. Now consider $\mathsf{pair}(m, V) = (x_k N_1 \cdots N_s, \{x_k : \mathsf{t}(k)\} \cup \Gamma_1 \cup \cdots \cup \Gamma_s)$, which follows from $(m, V) \rightsquigarrow (n_1, V), \ldots, (n_s, V) \rightsquigarrow^* \epsilon$ because $m := k\ n_1 \cdots n_s \in \mathsf{pre}(\tau)$ and $k \in V$, $(N_i, \Gamma_i) = \mathsf{pair}(n_i, V)$, for $1 \leq i \leq s\ (s \geq 0)$. By the induction hypothesis $\Gamma_i \vdash N_i : \mathsf{t}(n_i)$ and by definition $\{x_k : \mathsf{t}(k)\} \cup \Gamma_1 \cup \cdots \cup \Gamma_s$ is consistent. It follows from $m := k\ n_1 \cdots n_s \in \mathsf{pre}(\tau)$ that $\mathsf{t}(k) = \mathsf{t}(n_1) \rightarrow \cdots \rightarrow \mathsf{t}(n_s) \rightarrow \mathsf{t}(m)$. Therefore $\{x_k : \mathsf{t}(k)\} \cup \Gamma_1 \cup \cdots \cup \Gamma_s \vdash x_k N_1 \cdots N_s : \mathsf{t}(m)$. ◀

▶ **Example 19.** Consider α and $\mathsf{pre}(\alpha)$ from Example 8. Then,

$$(10, \emptyset) \quad \rightsquigarrow \quad (8, \{9\}) \rightsquigarrow (5, \{4, 9\}) \rightsquigarrow \epsilon. \qquad \text{Similarly,}$$

$$(10, \emptyset) \quad \rightsquigarrow \quad (8, \{9\}) \rightsquigarrow (6, \{9\}) \rightsquigarrow (1, \{0, 9\}) \rightsquigarrow (6, \{0, 9\}), (2, \{0, 9\})$$
$$\rightsquigarrow \quad (1, \{0, 9\}), (2, \{0, 9\}) \rightsquigarrow (2, \{0, 9\}) \rightsquigarrow \epsilon.$$

For the first two pairs in this last rewriting sequence we have respectively $\mathsf{pair}(10, \emptyset) = (\lambda x_9.x_9(\lambda x_0.x_9(\lambda x_0.x_0)x_0), \emptyset)$ and $\mathsf{pair}(8, \{9\}) = (x_9(\lambda x_0.x_9(\lambda x_0.x_0)x_0), \{x_9 : \mathsf{t}(9)\})$, where $\mathsf{t}(9) = (o \to o) \to o \to o$. Also, $\vdash \lambda x_9.x_9(\lambda x_0.x_9(\lambda x_0.x_0)x_0) : \mathsf{t}(10)$ and $\{x_9 : \mathsf{t}(9)\} \vdash x_9(\lambda x_0.x_9(\lambda x_0.x_0)x_0) : \mathsf{t}(8)$, where $\mathsf{t}(8) = o \to o$ and $\mathsf{t}(10) = \alpha$.

For the first rewriting sequence we have $\mathsf{pair}(10, \emptyset) = (\lambda x_9 x_4.x_4, \emptyset)$, $\mathsf{pair}(8, \{9\}) = (\lambda x_4.x_4, \emptyset)$, and $\mathsf{pair}(5, \{4, 9\}) = (x_4, \{x_4 : \mathsf{t}(4)\})$, where $\mathsf{t}(4) = o$. Also, $\vdash \lambda x_9 x_4.x_4 : \mathsf{t}(10)$, $\vdash \lambda x_4.x_4 : \mathsf{t}(8)$, and $\{x_4 : \mathsf{t}(4)\} \vdash x_4 : \mathsf{t}(5)$, with $\mathsf{t}(5) = o$.

▶ **Theorem 20.** $\mathsf{Nhabs}(\tau) \neq \emptyset$ if and only if $(\mathsf{n}(\tau), \emptyset) \rightsquigarrow^* \epsilon$.

Proof. The 'if' part follows from Lemma 18. For the 'only if' part, we show that for any term M, context Γ and type σ, if $\Gamma \vdash M : \sigma$, then $(\mathsf{n}(\sigma), V_\Gamma) \rightsquigarrow^* \epsilon$, where $V_\Gamma = \{ \mathsf{n}(\rho) \mid x : \rho \in \Gamma \}$. First, consider $M = \lambda x.N$ and suppose that we have $\Gamma \setminus \{x : \sigma_1\} \vdash \lambda x.N : \sigma_1 \to \sigma_2$, because $\Gamma \vdash N : \sigma_2$ and $\Gamma \cup \{x : \sigma_1\}$ is consistent. It follows from Lemma 4 that $\mathsf{st}(\lambda x.N) = \underline{\sigma_1 \to \sigma_2}$ is a positive occurrence of $\sigma_1 \to \sigma_2$ in τ. We consider the corresponding tuple $(\sigma_1 \to \sigma_2, m, k \to n) \in \mathsf{occT}(\tau)$, where $\mathsf{n}(\sigma_1 \to \sigma_2) = m$, $\mathsf{n}(\sigma_1) = k$, and $\mathsf{n}(\sigma_2) = n$. By Lemma 11, there is a rule $m := \lambda k.n \in \mathsf{pre}(\tau)$. Thus, $(m, V_{\Gamma \setminus \{x:\sigma_1\}}) \rightsquigarrow (n, V_{\Gamma \setminus \{x:\sigma_1\}} \cup \{k\})$. But, $V_{\Gamma \setminus \{x:\sigma_1\}} \cup \{k\} = V_\Gamma$ and $(n, V_\Gamma) \rightsquigarrow^* \epsilon$ follows from the induction hypothesis. Now, consider $M = xN_1 \cdots N_s$ and suppose that $\Gamma \vdash xN_1 \cdots N_s : \sigma$, because $\Gamma_i \vdash N_i : \sigma_i$, for $1 \leq i \leq s$ ($s \geq 0$), and because $\Gamma = \Gamma_1 \cup \cdots \cup \Gamma_s \cup \{x : \sigma_1 \to \cdots \to \sigma_s \to \sigma\}$ is consistent. By Lemma 4, $\mathsf{st}(x) = \underline{\sigma_1 \to \cdots \to \sigma_s \to \sigma}$ is a negative subpremise of τ. It follows from Lemma 11 that $m := k\,\mathsf{n}(\sigma_1) \cdots \mathsf{n}(\sigma_s) \in \mathsf{pre}(\tau)$, where $m = \mathsf{n}(\sigma)$ and $k = \mathsf{n}(\sigma_1 \to \cdots \to \sigma_s \to \sigma)$. Thus $(m, V_\Gamma) \rightsquigarrow (\mathsf{n}(\sigma_1), V_\Gamma), \ldots, (\mathsf{n}(\sigma_s), V_\Gamma)$. By the induction hypothesis, we have $(\mathsf{n}(\sigma_i), V_{\Gamma_i}) \rightsquigarrow^* \epsilon$, for $1 \leq i \leq s$. Since $V_{\Gamma_i} \subseteq V_\Gamma$, we conclude that $(\mathsf{n}(\sigma_1), V_\Gamma), \ldots, (\mathsf{n}(\sigma_s), V_\Gamma) \rightsquigarrow^* \epsilon$. ◀

4.3 Closure Properties

In this section we combine the pre-grammars of two types τ_1 and τ_2 in order to obtain pre-grammars for $\mathsf{Nhabs}(\tau_1) \cap \mathsf{Nhabs}(\tau_2)$ and for $\mathsf{Nhabs}(\tau_1) \cup \mathsf{Nhabs}(\tau_2)$, respectively. This allows us to extend our methods to a bigger range of types, such as sum types of rank 1.

▶ **Definition 21.** Given types τ_1 and τ_2, we define $\mathsf{N}(\tau_1 \cap \tau_2) = \mathsf{N}(\tau_1) \times \mathsf{N}(\tau_2)$. Furthermore, let $\mathsf{pre}(\tau_1 \cap \tau_2)$ denote the smallest set of rules satisfying the following.

- If $m_i := \lambda k_i.n_i \in \mathsf{pre}(\tau_i)$ $(i = 1, 2)$, then $(m_1, m_2) := \lambda(k_1, k_2).(n_1, n_2) \in \mathsf{pre}(\tau_1 \cap \tau_2)$;
- if $m_i := k_i\, n_1^i \cdots n_s^i \in \mathsf{pre}(\tau_i)$ for $i = 1, 2$ and $s \geq 0$, then $(m_1, m_2) := (k_1, k_2)\,(n_1^1, n_1^2) \cdots (n_s^1, n_s^2) \in \mathsf{pre}(\tau_1 \cap \tau_2)$.

▶ **Example 22.** For α from Example 2 and $\beta = ((a \to b) \to a \to b) \to (a \to b) \to a \to b$, pre-grammar $\mathsf{pre}(\beta)$ consists of the following rewriting rules.

14	:=	$\lambda 12.13$	11	:=	$\lambda 6.7 \mid 12\,8 \mid 10$	7	:=	$12\,8\,2 \mid 10\,4$	4 := $0 \mid 6$
13	:=	$\lambda 10.11 \mid 12$	8	:=	$\lambda 0.1 \mid 12\,8$	1	:=	$12\,8\,2 \mid 10\,4$	2 := $0 \mid 6$

After removing obsolete rules we obtain the following set of rules for $\mathsf{pre}(\alpha \cap \beta)$.

$(10, 14) := \lambda(9, 12).(8, 13) \qquad (8, 13) := \lambda(4, 10).(5, 11) \qquad (5, 11) := (4, 10)$

It is easy to see that a term M passes the type checking algorithm for this grammar if and only if $M \equiv_\alpha \lambda xy.y$, which is the only normal term that inhabits both types.

Note that the definition above can be extended in the obvious way to a finite number of intersections, i.e. types of the form $\tau_1 \cap \cdots \cap \tau_n$, for $n \geq 1$, where $\tau_1, \ldots, \tau_n$ are simple types. This corresponds to the set of intersection types of rank 1 [7]. We prove the correctness of our construction for the case of one intersection.

▶ **Theorem 23.** *Consider two simple types* τ_1, τ_2, *and a term* M. *Then, one has* $M \in$ $\mathsf{Nhabs}(\tau_1) \cap \mathsf{Nhabs}(\tau_2)$ *if and only if* $(M, (\mathsf{n}(\tau_1), \mathsf{n}(\tau_2)) \hookrightarrow^* \epsilon$, *with pre-grammar* $\mathsf{pre}(\tau_1 \cap \tau_2)$.

Proof. Consider two pairs (E_1, m_1) and (E_2, m_2) such that there is some λ-term Q and for $i = 1, 2$: there are placeholders $p_1^i, \ldots, p_r^i \in \mathsf{N}(\tau_i)$ and $\{x_1, \ldots, x_r\} \supseteq \mathsf{FV}(Q)$, such that $E_i = Q\theta_i$ for $\theta_i = [p_1^i/x_1, \ldots, p_r^i/x_r]$; and E_i is either of the form $\lambda x.(Q'\theta_i)$ or $p_j^i (Q_1\theta_i) \cdots (Q_n\theta_i)$. It follows from Definition 13 that, if some rule $\mathsf{r}_i \in \mathsf{pre}(\tau_i)$ applies to (E_i, m_i) $(i = 1, 2)$, then both pairs rewrite to a sequence of pairs of equal length, i.e. there is some $s \geq 0$ such that $(E_i, m_i) \hookrightarrow (E_1^i, m_1^i) \cdots (E_{s_i}^i, m_{s_i}^i)$ $(i = 1, 2)$, and for each $j = 1, \ldots, s$ we have that (E_j^1, m_j^1) and (E_j^2, m_j^2) verify the suppositions made on (E_1, m_1) and (E_2, m_2). Furthermore, this guarantees that $(\mathsf{r}_1, \mathsf{r}_2) \in \mathsf{pre}(\tau_1 \cap \tau_2)$, where $(\mathsf{r}_1, \mathsf{r}_2)$ denotes the rule in $\mathsf{pre}(\tau_1 \cap \tau_2)$, built from r_1 and r_2 as described in Definition 21.

If $M \in \mathsf{Nhabs}(\tau_1) \cap \mathsf{Nhabs}(\tau_2)$, then we have by Theorem 15 that $(M, \mathsf{n}(\tau_i)) \hookrightarrow^* \epsilon$, for $i = 1, 2$, in which pairs are assumed to be processed from left to right. The conditions above clearly apply to $(M, \mathsf{n}(\tau_1))$ and $(M, \mathsf{n}(\tau_2))$, and consequently to every other couple of pairs (E_1, m_1) and (E_2, m_2) in these rewriting sequences. One obtains a rewriting sequence for $(M, (\mathsf{n}(\tau_1), \mathsf{n}(\tau_2))) \hookrightarrow^* \epsilon$ using $Q(\theta_1, \theta_2)$ instead of $Q\theta_i$ $(i = 1, 2)$, where $(\theta_1, \theta_2) = [(p_1^1, p_1^2)/x_1, \ldots, (p_r^1, p_r^2)/x_r]$. The proof in the other direction is symmetrical, using the projections on the first or on the second coordinate in each step, in order to obtain rewriting sequences for $(M, \mathsf{n}(\tau_1)) \hookrightarrow^* \epsilon$ or for $(M, \mathsf{n}(\tau_2)) \hookrightarrow^* \epsilon$, respectively. ◀

In order to address sum types of rank 1 we will now define pre-grammars for the union of two languages. Consider rank 1 types τ_1 and τ_2, with sets $\mathsf{N}(\tau_1)$ and $\mathsf{N}(\tau_2)$ for which, without loss of generality, we assume we use two distinct sets of identifiers. Consequently, there is no overlapping of the corresponding grammars.

▶ **Definition 24.** Consider rank 1 types τ_1 and τ_2 and the corresponding identifiers $\mathsf{n}(\tau_i) \in$ $\mathsf{N}(\tau_i)$, for $i = 1, 2$. Let $\mathsf{N}(\tau_1 \cup \tau_2) = \{(\mathsf{n}(\tau_1), \mathsf{n}(\tau_2))\} \cup \mathsf{N}(\tau_1) \cup \mathsf{N}(\tau_2)$. Furthermore, consider the unique rule $\mathsf{n}(\tau_i) := \lambda k_i.n_i$ in $\mathsf{pre}(\tau_i)$ and let $\mathsf{pre}(\tau_i)' = \mathsf{pre}(\tau_i) \setminus \{\mathsf{n}(\tau_i) := \lambda k_i.n_i\}$, for $i = 1, 2$. We define,

$$\mathsf{pre}(\tau_1 + \tau_2) = \{(\mathsf{n}(\tau_1), \mathsf{n}(\tau_2)) := \lambda k_1.n_1; (\mathsf{n}(\tau_1), \mathsf{n}(\tau_2)) := \lambda k_2.n_2\} \cup \mathsf{pre}(\tau_1)' \cup \mathsf{pre}(\tau_2)'.$$

Again, it is straightforward to extend this definition to finite sums of rank 1 types.

▶ **Theorem 25.** *Consider two rank 1 types* τ_1, τ_2, *and a term* M. *Then, one has* $M \in$ $\mathsf{Nhabs}(\tau_1) \cup \mathsf{Nhabs}(\tau_2)$ *if and only if* $M \hookrightarrow^* \epsilon$, *with pre-grammar* $\mathsf{pre}(\tau_1 + \tau_2)$.

Proof. Straightforward, using Note 10 and Definition 24. ◀

5 Proving Inhabitation Related Problems to be in PSPACE

In this section we present the scheme of an alternating decision algorithm operating on tuples of the form (m, V, i), where $\{m\} \cup V \subseteq \mathsf{N}(\tau)$ and $i \in \mathbb{N}$. The algorithm takes as input a simple type τ, a positive integer depth, a vector/register reg, a function $\mathsf{f} : \mathbb{N} \times \mathsf{pre}(\tau) \times \mathsf{reg} \longrightarrow \mathsf{reg}$ manipulating the contents of reg depending on the values of $(i, \mathsf{r}) \in \mathbb{N} \times \mathsf{pre}(\tau)$, as well as an accepting condition $\mathsf{ac} : \mathsf{reg} \longrightarrow \{\top, \bot\}$. Functions f and ac are supposed to be computable

in linear time w.r.t. the size of their input. The integer depth is the limit for recursion, such that a loop of the algorithm aborts with failure, whenever this limit is exceeded. In each step, during the execution of the algorithm, one rule $r \in \mathsf{pre}(\tau)$ is applied to a tuple (m, V, i), and the values in reg are updated to $\mathsf{f}(i, r, \mathsf{reg})$. Upon a terminated run, condition $\mathsf{ac}(\mathsf{reg}_f)$ determines on success or failure, where reg_f is the configuration of reg at that point. We represent the empty register by $\varnothing$. Furthermore, let $\mathsf{f}_\varnothing$ be such that $\mathsf{f}_\varnothing(i, r, \mathsf{reg}) = \mathsf{reg}$ for all $(i, r) \in \mathbb{N} \times \mathsf{pre}(\tau)$, and $\mathsf{ac}_\top$ such that $\mathsf{ac}_\top(\mathsf{reg}) = \top$ for any configuration of reg. Depending on the instantiation of the parameters, the algorithm can be used to show that different inhabitation related problems, such as the emptiness problem, infiniteness, or principal inhabitation, are in PSPACE.

▶ **Definition 26** (PS). Consider a simple type τ, a positive integer depth, a register reg, as well as (linear) functions $\mathsf{f} : \mathbb{N} \times \mathsf{pre}(\tau) \times \mathsf{reg} \longrightarrow \mathsf{reg}$ and $\mathsf{ac} : \mathsf{reg} \longrightarrow \{\top, \bot\}$. Then, $\mathsf{PS}(\tau, \mathsf{depth}, \mathsf{reg}, \mathsf{f}, \mathsf{ac})$ operates as follows, starting with the initial tuple $(m, V, i) = (\mathsf{n}(\tau), \emptyset, 0)$:
- if $i > \mathsf{depth}$ the loop aborts with failure;
- otherwise the algorithm:
    - non-deterministically chooses a rule in $r \in \mathsf{pre}(\tau)$ such that:

    $$(m, V) \rightsquigarrow (n_1, V'), \ldots, (n_s, V');$$

    - updates reg according to $\mathsf{f}(i, r, \mathsf{reg})$;
    - universally applies to $(n_1, V', i+1), \ldots, (n_s, V', i+1)$.

A run is successful if $\mathsf{ac}(\mathsf{reg}_f) = \top$, where reg_f is the final configuration of reg.

Note that, other than by failure, a loop finishes if the rule chosen from $\mathsf{pre}(\tau)$ is such that $s = 0$. In order to show that PS is an alternating polynomial time algorithm w.r.t. the size $|\tau|$ of τ, we start by defining some measures on τ.

▶ **Definition 27.** Given a type τ, let $|\tau| = |\tau|_v + |\tau|_\rightarrow$, where $|\tau|_v$ represents the number of occurrences of type variables in τ and $|\tau|_\rightarrow$ the number of occurrences of $\rightarrow$ in τ. Furthermore, let $|\tau|^+$ and $|\tau|^-$ denote the number of positive occurrences of subformulas and the number of negative subpremises in τ, respectively. Similarly, we use $|\tau|_v^+$ and $|\tau|_v^-$ respectively for the number of positive and negative occurrences of type variables in τ.

The following lemma is a direct consequence of the definitions of $\mathsf{occT}(\tau)$, $\mathsf{N}(\tau)$, and $\mathsf{pre}(\tau)$.

▶ **Lemma 28.** *One has, $|\tau|^+ \leq |\tau|$, $|\tau|^- \leq |\tau|_\rightarrow < |\tau|$, as well as $|\mathsf{N}(\tau)| = |\tau|$. For the number of rules in $\mathsf{pre}(\tau)$ we have $|\mathsf{pre}(\tau)| \leq |\tau|^+ \cdot |\tau|^- + |\tau|_\rightarrow$. Furthermore, the number of elements of $\mathsf{N}(\tau)$ occurring in a rule of $\mathsf{pre}(\tau)$ is always $\leq |\tau|^+ + 1$.*

▶ **Example 29.** For $\alpha = ((o \rightarrow o) \rightarrow o \rightarrow o) \rightarrow o \rightarrow o$, we have $|\alpha| = |\alpha|_v + |\alpha|_\rightarrow = 6 + 5 = 11 = \mathsf{N}(\alpha)$. Furthermore, $|\alpha|^+ \cdot |\alpha|^- + |\alpha|_\rightarrow = 6 \cdot 3 + 5 = 23 \geq 14 = |\mathsf{pre}(\alpha)|$. Finally, the maximum number of identifiers occurring in the rules of $\mathsf{pre}(\alpha)$ is 4 and $4 \leq 6 + 1 = |\alpha|^+ + 1$.

▶ **Proposition 30.** Consider a type τ and constants $k_1, k_2 \in \mathbb{N}$. Suppose that $\mathsf{depth} \leq |\tau|^{k_1}$, $|\mathsf{reg}| \leq k_2 \cdot |\tau|$, and that functions $\mathsf{f} : \mathbb{N} \times \mathsf{pre}(\tau) \times \mathsf{reg} \longrightarrow \mathsf{reg}$ and $\mathsf{ac} : \mathsf{reg} \longrightarrow \{\top, \bot\}$ are computable in linear time w.r.t. the size of their input. Then, $\mathsf{PS}(\tau, \mathsf{depth}, \mathsf{reg}, \mathsf{f}, \mathsf{ac})$ is an alternating polynomial time algorithm w.r.t. $|\tau|$.

Proof. The algorithm is alternating by design. Polynomial time is a consequence of the conditions imposed on the complexity of depth, reg, f and ac. ◀

In the following we establish the relationship between a successful run of algorithm PS with $\mathsf{reg} = \varnothing$, $\mathsf{f}_\varnothing$ and $\mathsf{ac}_\top$, and the existence of a rewriting sequence for $(\mathsf{n}(\tau), \emptyset) \rightsquigarrow^* \epsilon$. Note, that each rewriting sequence of $(\mathsf{n}(\tau), \emptyset) \rightsquigarrow^* \epsilon$ can be represented in the usual way by a unique *derivation tree* t, whose internal nodes are labelled with pairs (m, V, i) and such that all leafs are labelled with ϵ. The root of t is $(\mathsf{N}(\tau), \emptyset, 0)$, and whenever a rule $\mathsf{r} \in \mathsf{pre}(\tau)$ is applied to a pair (m, V), such that $(m, V) \rightsquigarrow (n_1, V'), \ldots, (n_s, V')$, then the corresponding node in t, labelled with (m, V, i), has s children labelled with $(n_1, V', i+1), \ldots, (n_s, V', i+1)$ if $s > 0$, and it has one child labelled with ϵ if $s = 0$. Conversely, we can replace each pair (m, V) in the rewriting sequence by the label (m, V, i) of the corresponding node in t. Furthermore, the height of t is $\mathsf{height}(\mathsf{t}) = \mathsf{depth}(M)$, where $(M, \emptyset) = \mathsf{pair}(\mathsf{N}(\tau), \emptyset)$, corresponding to that rewriting sequence of $(\mathsf{n}(\tau), \emptyset) \rightsquigarrow^* \epsilon$.

▶ **Example 31.** The annotated version of the second rewriting sequence from Example 19 is as follows.

$$(10, \emptyset, 0) \quad \rightsquigarrow \quad (8, \{9\}, 1) \rightsquigarrow (6, \{9\}, 2) \rightsquigarrow (1, \{0, 9\}, 3) \rightsquigarrow (6, \{0, 9\}, 4), (2, \{0, 9\}, 4)$$
$$\rightsquigarrow \quad (1, \{0, 9\}, 5), (2, \{0, 9\}, 4) \rightsquigarrow (2, \{0, 9\}, 4) \rightsquigarrow \epsilon.$$

The corresponding derivation tree has height 6. Also, $\mathsf{depth}(\lambda x_9.x_9(\lambda x_0.x_9(\lambda x_0.x_0)x_0)) = 6$ and $\mathsf{pair}(10, \emptyset) = (\lambda x_9.x_9(\lambda x_0.x_9(\lambda x_0.x_0)x_0), \emptyset)$.

▶ **Lemma 32.** *Consider a type τ and an integer $d > 0$. Then, $\mathsf{PS}(\tau, d, \varnothing, \mathsf{f}_\varnothing, \mathsf{ac}_\top)$ succeeds if and only if there is a rewriting sequence for $(\mathsf{n}(\tau), \emptyset) \rightsquigarrow^* \epsilon$, whose derivation tree t has height $\leq d + 1$. Furthermore, $\mathsf{height}(\mathsf{t}) = \mathsf{depth}(M)$, where $(M, \emptyset) = \mathsf{pair}(\mathsf{n}(\tau), \emptyset)$ for that rewriting sequence of $(\mathsf{n}(\tau), \emptyset) \rightsquigarrow^* \epsilon$.*

Proof. It is easy to see that $\mathsf{PS}(\tau, d, \varnothing, \mathsf{f}_\varnothing, \mathsf{ac}_\top)$ succeeds if and only if there is some tree t with root $(\mathsf{n}(\tau), \emptyset, 0)$ and such that for every node (m, V, i) in that tree, there is a rule $\mathsf{r} \in \mathsf{pre}(\tau)$ such that $(m, V) \rightsquigarrow (n_1, V'), \ldots, (n_s, V')$, and node (m, V, i) has s children $(n_1, V', i+1), \ldots, (n_s, V', i+1)$ if $s > 0$, and one child labelled with ϵ if $s = 0$. The value of i in a node of t labelled with (m, V, i) is $\leq d$, and all leaf nodes are labelled with ϵ. Thus, the height of t is at most $d + 1$. On the other hand, every tree t satisfying the conditions above corresponds to an (annotated) rewriting sequence of $(\mathsf{n}(\tau), \emptyset) \rightsquigarrow^* \epsilon$ and vice-versa. It remains to show that $\mathsf{height}(\mathsf{t}) = \mathsf{depth}(M)$, where $(M, \emptyset) = \mathsf{pair}(\mathsf{n}(\tau), \emptyset)$ for that rewriting sequence of $(\mathsf{n}(\tau), \emptyset) \rightsquigarrow^* \epsilon$. Consider a subtree t' of t, whose root is labelled with a tuple (m, V, i), and the corresponding rewriting sequence of $(m, V) \rightsquigarrow^* \epsilon$. We show by induction on the height of this subtree that $\mathsf{height}(\mathsf{t}') = \mathsf{depth}(M)$, where $(M, \Gamma) = \mathsf{pair}(m, V)$. If $\mathsf{height}(\mathsf{t}') = 1$, then $(m, V) \rightsquigarrow \epsilon$ because $m := k \in \mathsf{pre}(\tau)$ and $k \in V$. Thus, $\mathsf{pair}(m, V) = (x_k, \{x_k : \mathsf{t}(k)\})$ and $\mathsf{depth}(x_k) = 1$. If (m, V, i) has $s > 0$ children labelled with $(n_1, V, i+1), \ldots, (n_s, V, i+1)$ because $m := k \ n_1 \cdots n_s \in \mathsf{pre}(\tau)$ and $k \in V$, then $(m, V) \rightsquigarrow (n_1, V), \ldots, (n_s, V)$ and $\mathsf{pair}(m, V) = (x_k \ N_1 \cdots N_s, \{x_k : \mathsf{t}(k)\} \cup \Gamma_1 \cup \cdots \cup \Gamma_s)$, where $(N_i, \Gamma_i) = \mathsf{pair}(n_i, V)$, for $1 \leq i \leq s$. Furthermore, $\mathsf{height}(\mathsf{t}')$ equals 1 plus the maximum of the heights of the subtrees rooted in $(n_1, V, i+1), \ldots, (n_s, V, i+1)$, while $\mathsf{depth}(x_k \ N_1 \cdots N_s)$ equals 1 plus the maximum of the depths of $N_1, \ldots, N_s$. Thus, the result follows from the induction hypothesis. Finally, suppose that (m, V, i) has one child labelled with $(n, V \cup \{k\}, i+1)$ because $m := \lambda k.n \in \mathsf{pre}(\tau)$. Then, $\mathsf{height}(\mathsf{t}')$ equals 1 plus the height of the subtree rooted in $(n, V \cup \{k\}, i+1)$. On the other hand, $\mathsf{pair}(m, V) = (\lambda x_k.N, \Gamma \setminus \{x_k : \mathsf{t}(k)\})$, where $(N, \Gamma) = \mathsf{pair}(n, V \cup \{k\})$. We have $\mathsf{depth}(\lambda x_k.N) = 1 + \mathsf{depth}(N)$ and consequently the result follows from the induction hypothesis. ◀

5.1   Emptiness

In the following we reprove the well-known result [14, 16], stating that the emptiness problem for TA_λ is in PSPACE, by instantiation of algorithm PS. We say that a derivation tree t corresponding to a particular rewriting sequence of $(\mathsf{N}(\tau), \emptyset) \rightsquigarrow^* \epsilon$ has a *repetition*, if and only if there is a branch in t containing two nodes with labels (m, V, i) and (m, V, i') such that $i \neq i'$. Furthermore, we have $(\mathsf{N}(\tau), \emptyset) \rightsquigarrow^* \epsilon$ if and only if there is some rewriting sequence for that fact, whose derivation tree t contains no repetition. By Lemma 32 it suffices to execute algorithm PS with a value for depth that guarantees that every derivation tree with $\mathsf{depth} > \mathsf{depth}$ has a repetition. For this, we define $\mathsf{D}(\tau) = |\tau|^+ \cdot |\tau|^-$.

▶ **Proposition 33.** $\mathsf{PS}(\tau, \mathsf{D}(\tau), \varnothing, \mathsf{f}_\varnothing, \mathsf{ac}_\top)$ *succeeds if and only if* $\mathsf{Nhabs}(\tau) \neq \emptyset$.

Proof. The limit $\mathsf{D}(\tau)$ is chosen so that for a pair (m, V, d) with $d > \mathsf{D}(\tau)$, there is a repetition in the corresponding derivation tree t. Since there are at most $|\tau|^+$ different identifiers m and at most $|\tau|^-$ different sets V, there has to be a repetition in the branch leading from the root of t to (m, V, d). Thus, the result follows from Lemma 32.   ◀

5.2   Counting

In [2], Ben-Yelles defined a counting algorithm that answers the question of how many normal inhabitants a given type τ has. The main focus, when asking this question, is usually on determining if $\mathsf{Nhabs}(\tau)$ is empty, finite or infinite. In [10], the infiniteness of $\mathsf{Nhabs}(\tau)$ was shown to be PSPACE complete. In the following, we show how algorithm PS can be instantiated in order to prove this problem to be in PSPACE.

We already argued that $\mathsf{Nhabs}(\tau) \neq \emptyset$ if and only if there is some derivation tree for $(\mathsf{N}(\tau), \emptyset) \rightsquigarrow^* \epsilon$ of height $\leq \mathsf{D}(\tau) + 1 = |\tau|^+ \cdot |\tau|^- + 1$. In the following we establish a lower limit $\mathsf{d}(\tau)$, such that the existence of a tree of height $> \mathsf{d}(\tau)$ guarantees that $|\mathsf{Nhabs}(\tau)| = \infty$. Consider a tree t containing a branch with two nodes $n = (m, V, d)$ and $n' = (m, V', d')$, with $d < d'$. Then, $V \subseteq V'$ and one can construct a new derivation tree by replacing in t the subtree $\mathsf{t}_{n'}$ rooted in n' by the subtree t_n rooted in n, changing every label (m'', V'', i) to $(m'', V'' \cup V', i + (d' - d))$. Repeating this process, it is possible to construct an infinite number of derivation trees of increasing height. Thus, $\mathsf{Nhabs}(\tau)$ is infinite. On the other hand, for $\mathsf{d}(\tau) = |\tau|^+$, if t has some branch of length $> \mathsf{d}(\tau)$, then this branch contains necessarily two such nodes n and n'. Now, suppose that the height of t is $\leq \mathsf{d}(\tau)$ and that some branch in t contains two nodes n and n' as above. Then $d, d' \leq \mathsf{d}(\tau)$ and $0 < (d' - d) < \mathsf{d}(\tau)$. Then, it is clear that repeating the process described above, at some point, one obtains a derivation tree of height D, with $\mathsf{d}(\tau) \leq D \leq \mathsf{D}(\tau)$, as long as $|\tau|^- > 1$. We conclude that for τ, such that $|\tau|^- > 1$, we have $\mathsf{Nhabs}(\tau) = \infty$ if and only if there is some derivation tree of height D, with $\mathsf{d}(\tau) + 1 \leq D \leq \mathsf{D}(\tau) + 1$.

▶ **Lemma 34.** *If* $|\tau|^- \leq 1$*, then* $\mathsf{Nhabs}(\tau) \neq \emptyset$ *iff* $\tau = a \rightarrow a$*, for which* $|\mathsf{Nhabs}(\tau)| = 1$.

Proof. If $|\tau|^- = 0$, then $\tau = a$ and $\mathsf{Nhabs}(\tau) = \emptyset$. For $|\tau|^- = 1$, it is easy to show, by induction on the number of implications in τ, that τ is of the form $(a_1 \rightarrow \cdots \rightarrow a_n \rightarrow b) \rightarrow a$, which is inhabited exactly if $n = 0$ and $a = b$.   ◀

▶ **Proposition 35.** *The counting problem for* $\mathsf{Nhabs}(\tau)$ *is in PSPACE.*

Proof. If $|\tau|^- \leq 1$, then $|\mathsf{Nhabs}(\tau)| = 1$ if $\tau = a \rightarrow a$, and $|\mathsf{Nhabs}(\tau)| = \emptyset$ otherwise.

If $|\tau|^- > 1$, then $|\mathsf{Nhabs}(\tau)| = \infty$ if and only if there is some derivation tree of height D such that $\mathsf{d}(\tau) < D \leq \mathsf{D}(\tau) + 1$. This can be checked by instantiating algorithm PS as follows:

- depth $= \mathsf{D}(\tau)$;
- $|\mathsf{reg}| = 1$ and $\mathsf{reg}[0] = 0$;
- $\mathsf{f}(i, \mathsf{r}, \mathsf{reg}) = (\text{IF } (i == \mathsf{d}(\tau)) \text{ THEN } \mathsf{reg}[0] := 1)$;
- $\mathsf{ac}(\mathsf{reg}) = (\mathsf{reg}[0] == 1)$.

If the algorithm succeeds, then $|\mathsf{Nhabs}(\tau)| = \infty$. Otherwise, according to Lemma 32 we can run $\mathsf{PS}(\tau, \mathsf{d}(\tau) - 1, \varnothing, \mathsf{f}_\varnothing, \mathsf{ac}_\top)$ in order to check if $\mathsf{Nhabs}(\tau)$ is finite, but not empty. ◀

5.3 Principal Inhabitation

We now use our algorithm to address the closely related problem of principal inhabitation, which although more complex, is still PSPACE-complete [6]. The *principal inhabitation problem* is about the existence of a normal inhabitant M of τ, such that τ is the principal type of M. A term M is a principal inhabitant of τ, if $\vdash M : \tau$ and if every type σ, such that $\vdash M : \sigma$, is an instance of τ. Then, τ is called the principal type of M. When searching for principal inhabitants, it is sufficient to consider principal inhabitants in long normal form, for which a characterisation was given in [4] in terms of proof trees, in the context of the formula-tree method. In this section we instantiate the algorithm PS to decide principal inhabitation, based on that characterisation. This characterisation was used in [1] to define deterministic principal inhabitation machines for normal inhabitants obtained from pre-grammars, following the formalism of Schubert et al. An inhabitant M of a type is called long, if every variable occurrence, which is in function position, is given as many arguments as allowed by its type. It is straightforward to change the definition of $\mathsf{pre}(\tau)$ in order to apply exactly to the set of long normal inhabitants of τ. For this, it is sufficient to drop in $\mathsf{pre}(\tau)$ all rules of the form $m := k\, n_1 \cdots n_s$ such that $\mathsf{lab}(m) \neq \mathsf{var}$. The pre-grammar thereby obtained is denoted by $\mathsf{preL}(\tau)$ and verifies the following. If $m^+ \in \mathsf{N}(\tau)$ and $\mathsf{lab}(m) = k \to n$, then there is exactly one rule for m in $\mathsf{preL}(\tau)$, which is $m := \lambda k.n$. If $m^+ \in \mathsf{N}(\tau)$ and $\mathsf{lab}(m) = \mathsf{var}$, then all rules for m are of the form $m := k\, n_1 \cdots n_s$ $(s \geq 0)$, such that $\mathsf{t}(k) = \mathsf{t}(n_1) \to \cdots \to \mathsf{t}(n_s) \to \mathsf{t}(n)$, where $\mathsf{lab}(n) = \mathsf{var}$ and $\mathsf{t}(m) = \mathsf{t}(n)$, i.e. m and n are different occurrences of the same type variable. For convenience we denote n by $\mathsf{tail}(k)$. Note that $\mathsf{tail}(k)$ is the root of the (unary) tree in graph $T(\tau)$, that contains k.

▶ **Example 36.** The pre-grammar $\mathsf{preL}(\alpha)$ for the set of long normal inhabitants of α from Example 2 is the following.

10	:=	$\lambda 9.8$	6	:=	$\lambda 0.1$	2	:=	$9\ 6\ 2 \mid 4 \mid 0$
8	:=	$\lambda 4.5$	5	:=	$9\ 6\ 2 \mid 4 \mid 0$	1	:=	$9\ 6\ 2 \mid 4 \mid 0$

The approach in [4] establishes that, initially all occurrences of type variables in τ have to be made different. Here, this is already achieved by the association of different identifiers to different occurrences of subtypes. During the search of an inhabitant, the application of a rule $m := k\, n_1 \cdots n_s$, as described above, forces that m and n must represent the same type variable in any type of that inhabitant[3]. When instantiating the algorithm, this information will be kept in register reg and the execution will only be successful if all occurrences of the same type variable are unified. The remaining condition in the characterisation of principal inhabitants in [4] states that all composed negative subpremises have to be used. This information will also be stored in reg. We denote the number of composed negative subpremises in τ by $|\tau|_c^-$ and define $\mathsf{P}(\tau) = |\tau|^+ \cdot |\tau|^- \cdot |\tau|_v \cdot |\tau|_c^-$ for the limit of recursion.

---

[3] Note that, limiting the search to long inhabitants avoids dealing with the unification of composed types, but restricts this operation to occurrences of type variables.

For practicality, we convention that type variables and negative subpremises have identifiers $0, \ldots, |\tau|_v - 1$, and $|\tau|_v, \ldots, |\tau|_v + |\tau|_c^- - 1$, respectively[4]. Finally, we denote by $\mathsf{var}(\tau)$ the number of different type variables in τ.

▶ **Example 37.** In our running example there are three negative subpremises, respectively with identifier 9, 4 and 0. We have $\mathsf{tail}(9) = 3$, $\mathsf{tail}(4) = 4$ and $\mathsf{tail}(0) = 0$. Only $\mathsf{t}(9)$ is composed. Thus, $|\alpha|_c^- = 1$ and $\mathsf{P}(\alpha) = 6 \cdot 3 \cdot 6 \cdot 1 = 108$, while $\mathsf{var}(\alpha) = 1$.

Now, we define a function $\mathsf{f_P}$ that stores the information concerning unification of different type variable occurrences and the use of composed negative subpremises in reg. For this, initially the identifier of each variable is stored in the first $|\tau|_v$ positions of reg, each representing its own class, which at that point is a singleton. The number of different classes, which is initially $|\tau|_v$, is stored in the last position of reg and decreased whenever two classes are merged. In this case, all elements (positions in reg) of these classes are represented by the same identifier. The intermediate positions of reg are used to register the application of composed negative subpremises.

```
f_P(i, r, reg):
IF r == (m := k n_1 ··· n_s) THEN
▪   n := tail(k);
▪   MIN := min(reg[m], reg[n]);
▪   MAX := max(reg[m], reg[n]);
▪   IF MIN ≠ MAX THEN
    ▪   reg[|τ|_v + |τ|_c^-] := reg[|τ|_v + |τ|_c^-] − 1;
    ▪   FOR (j = 0 TO |τ|_v − 1) DO
        *   IF (reg[j] == MAX) THEN reg[j] := MIN;
▪   IF (s > 0) THEN reg[k] := 1;
```

In order to determine success or failure of a run, function $\mathsf{ac_P}$ checks if all $|\tau|_c^-$ composed have been used and if there are exactly as many classes of occurrences of type variables as there are different type variables in τ.

```
ac_P(reg):
▪   COUNT := 0;
▪   FOR (j = |τ|_v TO |τ|_v + |τ|_c^- − 1) DO
    ▪   COUNT := COUNT + reg[j];
▪   IF (COUNT ≠ |τ|_c^-) THEN (RETURN ⊥)
    ELSE   (RETURN (reg[|τ|_v + |τ|_c^-] == var(τ)));
```

▶ **Proposition 38.** The principal inhabitation problem for $\mathsf{Nhabs}(\tau)$ is in PSPACE.

Proof. This can be checked by instantiating algorithm PS as follows:
▪ $\mathsf{depth} = \mathsf{P}(\tau)$;
▪ $|\mathsf{reg}| = |\tau|_v + |\tau|_c^- + 1$, $\mathsf{reg}[j] = j$ (for $0 \leq j \leq |\tau|_v - 1$),
  $\mathsf{reg}[j] = 0$ (for $|\tau|_v \leq j \leq |\tau|_v + |\tau|_c^- - 1$), and $\mathsf{reg}[|\tau|_v + |\tau|_c^-] = |\tau|_v$;
▪ $\mathsf{f} = \mathsf{f_P}$ and $\mathsf{ac} = \mathsf{ac_P}$.
Function $\mathsf{f_P}$ registers (during the execution of PS) all necessary information for deciding on principality in reg, which is checked by $\mathsf{ac_P}$ after completion of a run. Thus, there is

---

[4] This convention does not hold for the composed negative subpremise with identifier 9 in our example.

some principal inhabitant of τ iff there is a successful run using a limit of recursion, possibly bigger than $\mathsf{depth} = \mathsf{P}(\tau)$. We consider such a successful run (for a principal inhabitant), and the corresponding derivation tree t. Finally, we argue that it is possible to obtain a new derivation tree from t, corresponding to a successful run, within the established limit $\mathsf{P}(\tau)$. Consider any node $n = (m, V, i)$ in t. We associate to node n the number Eq_n of equivalence classes, as well as the set C_n of negative composed subpremises, that are induced by the derivation steps in the subtree rooted in n. There is a repetition in a branch of t if it contains two nodes $n = (m, V, i)$ and $n' = (m, V, i')$ with $i < i'$, such that $Eq_n = Eq_{n'}$ and $C_n = C_{n'}$. If that is the case, one can replace the subtree rooted in n by the smaller subtree rooted in n', obtaining a tree still corresponding to a successful run. Since $i < i'$ implies that $Eq_n \leq Eq_{n'} \leq |\tau|_v$, as well as $|\tau|_c^- \geq |C_n| \geq |C_{n'}|$, there is a repetition in every branch of length $\geq \mathsf{P}(\tau)$. Consequently, the process described above can be repeated until one obtains a derivation tree, thus a successful run, within the limit established for depth. ◄

6 Conclusions

In this paper we presented a unifying framework to study type inhabitation related problems and their complexity, using the notion of pre-grammar. From the pre-grammar of a type we obtained different methods to address several inhabitation related problems. A scheme for a decision algorithm was given, which we instantiated to decide emptiness, counting and principal inhabitation. Since each instantiation produces a polynomial time alternating algorithm, this also shows these problems to be in PSPACE. For principal inhabitants we focused on terms in long normal form, for which we used a simplified and smaller set of rules. In a similar way, one could define different sets of pre-grammar rules, corresponding to particular subclasses of terms, such as terms in total discharge form, term-schemes, etc. This is left for future work, where we also would like to further develop the study of closure properties, in particular study an instantiation of our algorithm for union types of rank 1.

References

1   Sandra Alves and Sabine Broda. Inhabitation machines: determinism and principality. In *Applications, NCMA 2017*, pages 57–70, 2017.
2   Ch. Ben-Yelles. *Type Assignment in the Lambda-Calculus: Syntax and Semantics.* PhD thesis, University College of Swansea, September 1979.
3   S. Broda and L. Damas. Counting a type's (principal) inhabitants. *Fundam. Inform.*, 45(1-2):33–51, 2001.
4   S. Broda and L. Damas. On long normal inhabitants of a type. *J. Log. and Comput.*, 15:353–390, June 2005.
5   M.W. Bunder. Proof finding algorithms for implicational logics. *Theoretical Computer Science*, 232(1–2):165–186, 2000.
6   Andrej Dudenhefner and Jakob Rehof. The complexity of principal inhabitation. In *Computation and Deduction, FSCD 2017*, volume 84 of *LIPIcs*, pages 15:1–15:14, 2017.
7   Silvia Ghilezan. Inhabitation in intersection and union type assignment systems. *J. Log. Comput.*, 3(6):671–685, 1993.
8   J.R. Hindley. *Basic Simple Type Theory*, volume 42 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1997.
9   R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc*, 146:29–60, December 1969.
10   S. Hirokawa. Infiniteness of proof (α) is polynomial-space complete. *Theor. Comput. Sci.*, 206(1-2):331–339, 1998.

11   W.A. Howard. The formulas-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980.

12   Y. Komori and S. Hirokawa. The number of proofs for a BCK-formula. *J. Symb. Log.*, 58(2):626–628, 1993.

13   Aleksy Schubert, Wil Dekkers, and Hendrik Pieter Barendregt. Automata theoretic account of proof search. In *CSL 2015*, pages 128–143, 2015.

14   R. Statman. Intuitionistic propositional logic is polynomial-space complete. *Theor. Comput. Sci.*, 9:67–72, 1979.

15   M. Takahashi, Y. Akama, and S. Hirokawa. Normal proofs and their grammar. *Information and Computation*, 125(2):144–153, 1996.

16   P. Urzyczyn. Inhabitation in typed lambda-calculi (a syntactic approach). In *TLCA'97*, volume 1210 of *LNCS*, pages 373–389. Springer, 1997.

Confluence of Prefix-Constrained Rewrite Systems

Nirina Andrianarivelo
LIFO - Université d'Orléans, B.P. 6759, 45067 Orléans cedex 2, France
Nirina.Andrianarivelo@univ-orleans.fr

Pierre Réty
LIFO - Université d'Orléans, B.P. 6759, 45067 Orléans cedex 2, France
Pierre.Rety@univ-orleans.fr

—————— **Abstract** ——————

Prefix-constrained rewriting is a strict extension of context-sensitive rewriting. We study the confluence of prefix-constrained rewrite systems, which are composed of rules of the form $L : l \rightarrow r$ where L is a regular string language that defines the allowed rewritable positions. The usual notion of Knuth-Bendix's critical pair needs to be extended using regular string languages, and the convergence of all critical pairs is not enough to ensure local confluence. Thanks to an additional restriction we get local confluence, and then confluence for terminating systems, which makes the word problem decidable. Moreover we present an extended Knuth-Bendix completion procedure, to transform a non-confluent prefix-constrained rewrite system into a confluent one.

1 Introduction

Term rewriting is a rule-based formalism that can be used to study properties of functional programs, security protocols, musical rhythmics,... More generally, it provides a finite abstraction of a system whose configurations are represented by ranked terms. In this framework, and also to ensure the termination of rewrite computations, it is often necessary to restrict the possible rewrite positions, using strategies, or by allowing only some redex positions. In context-sensitive rewriting [10], some arguments of a function symbol may be defined as being non-rewritable. Prefix-constrained rewriting [8] is an extension of context-sensitive rewriting, where rewritable positions are defined by a regular string language that indicates the allowed prefixes.

Given a term t, a normal form of t is an irreducible term (denoted $t{\downarrow}$) obtained by rewriting t. Termination of a rewrite relation $\rightarrow_R$ ensures the existence of normal forms, whereas confluence ensures their uniqueness. Together, termination and confluence ensure that the word problem is decidable, because $t =_R t'$ is equivalent to $t{\downarrow} = t'{\downarrow}$. On the other hand, from a functional programming point of view, termination ensures that any program run will terminate, and confluence ensures that all functions are deterministic, i.e. each function call yields at most one result. These properties have also been addressed for context-sensitive rewriting ([5] for termination and [11] for confluence). On the other hand, the termination of prefix-constrained rewriting has been addressed in [1]. Both [5] and [1] consist in transforming the context-sensitive or prefix-constrained rewrite system into an ordinary one by a termination-preserving transformation, and studying the termination of the ordinary rewrite system.

In this paper, we study the confluence of prefix-constrained rewrite systems. In contrast to ordinary rewriting, prefix-constrained rewriting (and context-sensitive rewriting) is not closed under context application, which is a major difference. This is why the usual notion of Knuth-Bendix's critical pair needs to be extended (using regular string languages), and the convergence of all critical pairs is not enough to ensure local confluence. Thanks to an additional restriction we get local confluence, and then confluence for terminating systems, which makes the word problem decidable. Moreover we present an extended Knuth-Bendix completion procedure, to transform a non-confluent prefix-constrained rewrite system into a confluent one.

The paper is organized as follows. The preliminaries are introduced in Section 2. Local-confluence is studied in Section 3, and a comparison with [11] is given at the end of the section. The operational point of view to handle string languages is given in Section 4. An extended Knuth-Bendix completion procedure is presented in Section 5. Further work is outlined in Section 6.

2 Preliminaries

Term and Substitution. Consider a *finite ranked alphabet* Σ and a set of variables X. Each symbol $f \in \Sigma$ has a unique arity, denoted by $ar(f)$. The notions of *first-order term*, *position* and *substitution* are defined as usual. $T(\Sigma, X)$ denotes the set of terms over $\Sigma \cup X$, and $T(\Sigma)$ denotes the set of ground terms (without variables) over Σ. For a term t, $Var(t)$ is the set of variables of t, $Pos(t)$ is the set of positions of t, $PosVar(t)$ is the set of variable positions of t, $PosNonVar(t) = Pos(t) \backslash PosVar(t)$, and ϵ is the root position. For $p \in Pos(t)$, $t(p)$ is the symbol of $\Sigma \cup X$ occurring at position p in t, and $t|_p$ is the subterm of t at position p. For $p, p' \in Pos(t)$, $p < p'$ means that p occurs in t strictly above p', whereas $p \parallel p'$ means that $p \neq p'$ and $p \not< p'$ and $p' \not< p$. The term t is *linear* if each variable of t occurs only once in t. The term $t[t']_p$ is obtained from t by replacing the subterm at position p by t'.

Given σ and σ' two substitutions, $\sigma \circ \sigma'$ denotes the substitution such that for all variable x, $\sigma \circ \sigma'(x) = \sigma(\sigma'(x))$. The substitution σ is a *unifier* of the terms t and t' if $\sigma(t) = \sigma(t')$. If in addition, for all unifier θ of t and t', there exists a substitution γ such that $\theta = \gamma \circ \sigma$, then σ is called the *most general unifier* of t and t' (denoted $mgu(t, t')$). If it exists, the most general unifier is unique up to a variable renaming.

Term Rewrite System (TRS). A *rewrite rule* is an oriented pair of terms, written $l \to r$. We always assume that l is not a variable, and $Var(r) \subseteq Var(l)$. A *rewrite system* R is a finite set of rewrite rules. *lhs* stands for left-hand-side, *rhs* for right-hand-side. The *rewrite relation* $\to_R$ is defined as follows: $t \to_R t'$ if there exist a non-variable position $p \in Pos(t)$, a rule $l \to r \in R$, and a substitution θ s.t. $t|_p = \theta(l)$ and $t' = t[\theta(r)]_p$ (also denoted $t \to_R^p t'$). $\to_R^+$ denotes the transitive closure of $\to_R$, and $\to_R^*$ denotes the reflexive-transitive closure of $\to_R$. t' is a *descendant* of t if $t \to_R^* t'$. If I is a set of ground terms, $R^*(I)$ denotes the set of descendants of elements of I. The rewrite rule $l \to r$ is *left (resp. right) linear* if l (resp. r) is linear. R is *left (resp. right) linear* if all its rewrite rules are left (resp. right) linear. R is *linear* if R is both left and right linear. $l \to r$ is said *collapsing* if r is a variable.

Let $l_1 \to r_1$ and $l_2 \to r_2$ be rewrite rules such that $l_1|_p$ and l_2 are unifiable for some $p \in PosNonVar(l_1)$. Let $\sigma = mgu(l_1|_p, l_2)$. Then the pair of terms $(\sigma(r_1), \sigma(l_1)[\sigma(r_2)]_p)$ is called *critical pair*[1].

---

[1] As usual, we do not consider trivial critical pairs $(\sigma(r_1), \sigma(r_1))$ coming from the case where $l_1 = l_2$ and

Context-Sensitive Term Rewrite System (CS-TRS) [4, 10]. A *context-sensitive rewrite relation* is a sub-relation of the ordinary rewrite relation in which rewritable positions are indicated by specifying arguments of function symbols. A mapping $\mu : \Sigma \to P(\mathbb{N})$ is said to be a *replacement map* (or Σ-map) if $\mu(f) \subseteq \{1, \ldots, ar(f)\}$ for all $f \in \Sigma$. A *context-sensitive* term rewriting system (CS-TRS) is a pair $\mathcal{R} = (R, \mu)$ composed of a TRS and a replacement map. The set of μ-replacing positions[2] $Pos^\mu(t)$ ($\subseteq Pos(t)$) is recursively defined: $Pos^\mu(t) = \{\epsilon\}$ if t is a constant or a variable, otherwise $Pos^\mu(f(t_1, \ldots, t_n)) = \{\epsilon\} \cup \{i.p \mid i \in \mu(f), p \in Pos^\mu(t_i)\}$. The rewrite relation induced by a CS-TRS $\mathcal{R}$ is defined: $t \hookrightarrow_{\mathcal{R}} t'$ if $t \to_R^p t'$ for some $p \in Pos^\mu(t)$.

▶ **Example 1.** Let $\Sigma = \{f^{\backslash 2}, g^{\backslash 2}, a^{\backslash 0}, b^{\backslash 0}\}$ and $R = \{a \to b\}$ with $\mu(f) = \{1\}$ and $\mu(g) = \{2\}$. The positions allowed by μ in the term $\boldsymbol{f}(\boldsymbol{a}, a)$ are written in bold. Then the only derivation issued from this term is $f(a, a) \hookrightarrow_{\mathcal{R}} f(b, a)$. On the other hand, consider $t = \boldsymbol{f}(\boldsymbol{g}(a, \boldsymbol{a}), a)$. Then the only derivation issued from this term is $f(g(a, a), a) \hookrightarrow_{\mathcal{R}} f(g(a, b), a)$.

String Language. Given an alphabet Σ, the set of all strings over Σ is denoted by Σ^*, and ϵ denotes the *empty string*. Symbol '.' denotes the concatenation.

String Automaton. A finite *string automaton* is a 5-tuple $\mathcal{A} = (\Sigma, Q, Q_I, Q_f, \Delta)$ where Q is a set of states, $Q_I \subseteq Q$ is the set of initial states, $Q_f \subseteq Q$ is the set of final states, and $\Delta \subseteq Q \times \Sigma \times Q$ is the set of *transitions*. The *transition relation* $\mapsto_\Delta$ between elements of $Q \times \Sigma^*$ is defined as follows: for $q, q' \in Q$, $a \in \Sigma$, $w \in \Sigma^*$, $(q, a.w) \mapsto_\Delta (q', w)$ iff $(q, a, q') \in \Delta$. The reflexive-transitive closure of $\mapsto_\Delta$ is written $\mapsto_\Delta^*$. The language recognized by $\mathcal{A}$ is $L_\mathcal{A} = \{w \in \Sigma^* \mid \exists q_I \in Q_I, \exists q_f \in Q_f, (q_I, w) \mapsto_\Delta^* (q_f, \epsilon)\}$. A *regular string language* is a set of strings recognized be some finite string automaton. It is well known that regular languages are closed under union, intersection, complement, and membership and emptiness are decidable.

$\mathcal{A}$ is said *deterministic* (resp. *complete*) if Q_I contains at most (resp. at least) one state, and for each $q \in Q$ and $a \in \Sigma$, there exists at most (resp. at least) one $q' \in Q$ such that $(q, a, q') \in \Delta$. It is well known that every automaton can be determinized and completed into an automaton that recognizes the same language. However the determinization step is exponential in the number of states. Let us write $\bar{\mathcal{A}} = (\Sigma, Q, Q_I, Q \backslash Q_f, \Delta)$. If $\mathcal{A}$ is deterministic and complete, it is well known that $\bar{\mathcal{A}}$ is deterministic and complete, and $L_{\bar{\mathcal{A}}} = \Sigma^* \backslash L_\mathcal{A}$, i.e. $\bar{\mathcal{A}}$ recognizes the complement of the language of $\mathcal{A}$.

Consider the automata $\mathcal{A}_1 = (\Sigma, Q^1, Q_I^1, Q_f^1, \Delta^1)$ and $\mathcal{A}_2 = (\Sigma, Q^2, Q_I^2, Q_f^2, \Delta^2)$. Let us define the automaton $\mathcal{A}_1 \cap \mathcal{A}_2 = (\Sigma, Q^1 \times Q^2, Q_I^1 \times Q_I^2, Q_f^1 \times Q_f^2, \Delta^1 \otimes \Delta^2)$ with $\Delta^1 \otimes \Delta^2 = \{((q_1, q_2), a, (q_1', q_2')) \mid (q_1, a, q_1') \in \Delta_1 \wedge (q_2, a, q_2') \in \Delta_2\}$. It is well known that $L_{\mathcal{A}_1 \cap \mathcal{A}_2} = L_{\mathcal{A}_1} \cap L_{\mathcal{A}_2}$, i.e. the automaton $\mathcal{A}_1 \cap \mathcal{A}_2$ recognizes the language intersection. Moreover if $\mathcal{A}_1$ and $\mathcal{A}_2$ are deterministic and complete, so is $\mathcal{A}_1 \cap \mathcal{A}_2$.

Prefix Constrained Term Rewrite System (pCTRS) [8]. Prefix constrained rewriting allows rewrite steps only at the positions p of t s.t. the path from the root of t and p belongs to a given regular string language. More precisely, consider the set of directions $Dir(\Sigma) = \{\langle g, i \rangle \mid g \in \Sigma, 1 \leq i \leq ar(g)\}$. For a variable $x \in X$, let $path(x, \epsilon) = \epsilon$ and for

---

$r_1 = r_2$ and $p = \epsilon$.

[2] Also called positions allowed by μ.

a term $t = g(t_1, \ldots, t_{(ar(g))}) \in T(\Sigma, X)$ and a position p, $path(t, p) \in Dir(\Sigma)^*$ is defined recursively by:

$$path(g(t_1, \ldots, t_{(ar(g))}), \epsilon) = \epsilon$$
$$path(g(t_1, \ldots, t_{(ar(g))}), i.p) = \langle g, i \rangle.path(t_i, p) \ \ with \ 1 \leq i \leq ar(g) \ and \ i.p \in Pos(t)$$

A *prefix constrained rewrite system* is a finite set R of prefix constrained rewrite rules of the form $L : l \to r$ s.t. $L \subseteq Dir(\Sigma)^*$ is a regular string language over $Dir(\Sigma)$, $l \in T(\Sigma, \mathcal{X}) \backslash \mathcal{X}$, and $r \in T(\Sigma, var(l))$. A term t is rewritten to t' in one step by a pCTRS R, denoted by $t \hookrightarrow_R t'$, if there exist a prefix-constrained rewrite rule $L : l \to r$ in R, a position $p \in Pos(t)$ s.t. $path(t, p) \in L$, and a substitution σ s.t. $t|_p = \sigma(l)$ and $t' = t[\sigma(r)]_p$. The reflexive-transitive closure of $\hookrightarrow_R$ is denoted by $\hookrightarrow_R^*$. The equality $=_R$ is the reflexive, symmetric and transitive closure of the pCTRS rewriting $\hookrightarrow_R$. A rewrite step $t \hookrightarrow_R t'$ at position p of t by rewrite rule $L : l \to r$ through substitution σ is noted $t \hookrightarrow_{[p, L:l \to r, \sigma]} t'$. Let us note that prefix-constrained rewriting is stable under instantiation.

▶ **Example 2.** Let $\Sigma = \{f^{\backslash 2}, g^{\backslash 2}, a^{\backslash 0}, b^{\backslash 0}\}$ and $R = \{(\langle f, 1 \rangle. \langle g, 2 \rangle)^* : a \to b\}$. Let $t = \boldsymbol{f}(g(a, \boldsymbol{a}), a)$. Note that $t(1.2) = a$ (in bold in t) and $path(t, 1.2) = \langle f, 1 \rangle. \langle g, 2 \rangle \in (\langle f, 1 \rangle. \langle g, 2 \rangle)^*$. Then this position can be reduced by prefix constrained rewriting, i.e. $t = \boldsymbol{f}(g(a, \boldsymbol{a}), a) \hookrightarrow_R \boldsymbol{f}(g(a, b), a)$, whereas the other occurrences of a are not reducible.
Note that the term $f(a, a)$ is not reducible by the pCTRS R, whereas it is reducible by the CS-TRS of Example 1. However the pCTRS $R_1 = \{(\langle f, 1 \rangle | \langle g, 2 \rangle)^* : a \to b\}$ is equivalent to the CS-TRS of Example 1.

▶ Remark. Context-sensitive rewriting is a particular case of prefix-constrained rewriting [8].

Confluence and Church-Rosser Property. For any binary relation S over the set of terms, let S^* be the reflexive-transitive closure of S, and $=_S$ be the reflexive-symmetric-transitive closure of S.
We say that the pair of terms (t_1, t_2) *converges* for S (denoted $t_1 \downarrow_S t_2$) if there exists a term t' such that $t_1 S^* t'$ and $t_2 S^* t'$.
S is said *locally confluent* if $t S t_1$ and $t S t_2$ implies $t_1 \downarrow_S t_2$, for all terms t, t_1, t_2.
S is said *confluent* if $t S^* t_1$ and $t S^* t_2$ implies $t_1 \downarrow_S t_2$, for all terms t, t_1, t_2.
S has the *Church-Rosser property* if $t_1 =_S t_2$ implies $t_1 \downarrow_S t_2$, for all terms t, t_1, t_2.

▶ **Theorem 3.** *[3] Church-Rosser property and confluence are equivalent.*

S is said *terminating* (or *well-founded*) if there is no infinite sequence of terms $t_1 S t_2 S t_3 S \ldots$.

▶ **Theorem 4.** *(Newman's lemma) [6] If S is locally confluent and terminating, then S is confluent.*

Now, let us consider a TRS R and the associated binary relation $\to_R$.

▶ **Theorem 5.** *(Knuth-Bendix's theorem) [9] R is locally confluent[3] if and only if all critical pairs of R are convergent.*

---

[3] I.e. $\to_R$ is locally confluent.

3 Local Confluence of pCTRSs

When positions p_1 and p_2 are parallel, rewriting at p_1 does not change the prefix of p_2, and conversely. Therefore such a peak converges as for ordinary TRSs.

▶ **Lemma 6.** *Let $R = \{L_1 : l_1 \rightarrow r_1, L_2 : l_2 \rightarrow r_2\} \cup R'$ be a pCTRS.*
If $t \hookrightarrow_{[p_1, L_1 : l_1 \rightarrow r_1, \sigma_1]} t_1$ and $t \hookrightarrow_{[p_2, L_2 : l_2 \rightarrow r_2, \sigma_2]} t_2$ and $p_1 \parallel p_2$,
then $t_1 \hookrightarrow_{[p_2, L_2 : l_2 \rightarrow r_2, \sigma_2]} t_3$ and $t_2 \hookrightarrow_{[p_1, L_1 : l_1 \rightarrow r_1, \sigma_1]} t_3$.

Proof. Since $p_1 \parallel p_2$, we have $t_1|_{p_2} = t|_{p_2}$ and $path(t_1, p_2) = path(t, p_2) \in L_2$. Then $t_1 \hookrightarrow t_3$. Since $p_1 \parallel p_2$, we have $t_2|_{p_1} = t|_{p_1}$ and $path(t_1, p_1) = path(t, p_1) \in L_1$. Then $t_2 \hookrightarrow t_3$. ◀

With an ordinary TRS, a peak coming from an overlap in a variable position converges. It may be wrong when considering a pCTRS. Consequently, a pCTRS without critical pairs may not be locally confluent.

▶ **Example 7.** Consider the pCTRS $R = \{\{\epsilon\} : f(x) \rightarrow g(x), \{\langle f, 1 \rangle\} : a \rightarrow b\}$. So $f(a) \hookrightarrow_R g(a)$ and $f(a) \hookrightarrow_R f(b) \hookrightarrow_R g(b)$. Note that $g(a)$ is irreducible by R because the second rewrite rule needs a prefix with symbol f to be applied. Then this peak starting from $f(a)$ does not converge, therefore R is not locally confluent. Moreover R does not have critical pairs.

In the previous example, the non-convergent peak comes from the fact that along the step $f(a) \hookrightarrow_R g(a)$, the occurrence of a in $f(a)$ is allowed to be reduced by the second rule, whereas it is forbidden in $g(a)$. In other words, the prefix $\langle f, 1 \rangle$ of a in $f(a)$ belongs to the language of the second rule, whereas the prefix $\langle g, 1 \rangle$ of a in $g(a)$ does not. We introduce the notion of *prefix-preserving* to avoid this situation, which is based on the same idea as in the context-sensitive case (Definition 4.4 of [11]).

Notations: for a variable x and a term t, let $Pos(t, x) = \{p \in Pos(t) \mid t(p) = x\}$. On the other hand, we use the character '.' to denote the string concatenation.

▶ **Definition 8.** The pCTRS R is *prefix-preserving* if for all rewrite rules $L_1 : l_1 \rightarrow r_1$ and $L_2 : l_2 \rightarrow r_2$ of R, for all $x \in Var(l_1)$, for all $p, p' \in Pos(l_1, x)$, for all $p'' \in Pos(r_1, x)$, for all $u, w \in Dir(\Sigma)^*$:

$$u \in L_1 \land u.path(l_1, p).w \in L_2 \implies u.path(l_1, p').w \in L_2 \land u.path(r_1, p'').w \in L_2$$

In the previous definition, p' is for considering the case where l_1 is not linear. The pCTRS of Example 7 is not prefix-preserving (with $u = w = \epsilon$).

▶ **Example 9.** Consider the pCTRS $R = \{L : if(true, x, y) \rightarrow x, L : if(false, x, y) \rightarrow y\}$, where L is the set of all words of $Dir(\Sigma)^*$ except the words that contain at least one occurrence of $\langle if, 2 \rangle$ or $\langle if, 3 \rangle$. Thus, the first argument (the condition) of if should be evaluated before the second or the third argument. R is prefix-preserving because the position of x (resp. y) in the left-hand-side of the first (resp. second) rule is forbidden with respect to L, i.e. the pre-condition of the implication of Definition 8, that means more precisely, $\forall u \in Dir(\Sigma)^*, \forall w \in Dir(\Sigma)^*$ $u.path(l, pos(l, x)).w \in L$, is always wrong.

▶ **Lemma 10.** *Let $R = \{L_1 : l_1 \rightarrow r_1, L_2 : l_2 \rightarrow r_2\} \cup R'$ be a prefix-preserving pCTRS.*
If $t \hookrightarrow_{[p_1, L_1 : l_1 \rightarrow r_1, \sigma_1]} t_1$ and $t \hookrightarrow_{[p_2, L_2 : l_2 \rightarrow r_2, \sigma_2]} t_2$ and $p_2 = p_1.v.w$ with $v \in Pos(l_1, x)$ for some variable x, then there exist t_3 and t_4 such that $t_1 \hookrightarrow^*_{[p_1.v'_1.w, \ldots, p_1.v'_m.w, L_2 : l_2 \rightarrow r_2, \sigma_2]} t_4$ and $t_2 \hookrightarrow^*_{[p_1.v_1.w, \ldots, p_1.v_n.w, L_2 : l_2 \rightarrow r_2, \sigma_2]} t_3 \hookrightarrow_{[p_1, L_1 : l_1 \rightarrow r_1, \sigma'_1]} t_4$, with $Pos(r_1, x) = \{v'_1, \ldots, v'_m\}$ and $Pos(l_1, x) = \{v, v_1, \ldots, v_n\}$.

Proof. Let us write $u = path(t, p_1)$. Then we have $u \in L_1$ and $path(t, p_1.v.w) \in L_2$. But $path(t, p_2) = path(t_1, p_1.v.w) = u.path(l_1, v).path(\sigma_1(x), w) \in L_2$.

Since R is prefix-preserving, we have $\forall i, u.path(l_1, v_i).path(\sigma_1(x), w) \in L_2$, then $t_2 \to^* t_3$ and $u.path(r_1, v_i').path(\sigma_1(x), w) \in L_2$, then $t_1 \to^* t_4$.

Furthermore, $path(t_3, p_1) = path(t, p_1) \in L_1$ then $t_3 \hookrightarrow_{[p_1, L_1: l_1 \to r_1, \sigma_1']} t_4$ with $\sigma_1'(x) = \sigma_1(x)[\sigma_2(r_2)]_w$ and $\forall y$, s.t. $y \neq x, \sigma_1'(y) = \sigma_1(y)$. ◀

Prefix-preserving is helpful to get local confluence, but it is not always necessary. The following pCTRS is locally confluent, whereas it is not prefix-preserving.

▶ **Example 11.**

$$R = \{\{\epsilon\} : f(x) \xrightarrow{1} g(x), \ \{\langle f, 1 \rangle . \langle h, 1 \rangle\} : a \xrightarrow{2} b, \ \{\langle g, 1 \rangle\} : h(a) \xrightarrow{3} h(b)\}$$

The only peak is $f(h(a)) \hookrightarrow g(h(a))$ by rule 1, and $f(h(a)) \hookrightarrow f(h(b))$ by rule 2. This peak is convergent since $g(h(a)) \hookrightarrow g(h(b))$ by rule 3 and $f(h(b)) \hookrightarrow g(h(b))$ by rule 1. Therefore R is locally confluent, and consequently confluent since R is terminating. Note the use of rule number 3 to get confluence.

Let L_1 and L_2 be the prefix-languages of rules 1 and 2 respectively. R is not prefix-preserving because using the notations of Definition 8, let $u = \epsilon \in L_1$ and $w = \langle h, 1 \rangle$, and we have $u.path(f(x), 1).w = \langle f, 1 \rangle . \langle h, 1 \rangle \in L_2$ whereas $u.path(g(x), 1).w = \langle g, 1 \rangle . \langle h, 1 \rangle \notin L_2$.

If we replace rule 3 by $\{\langle g, 1 \rangle\} : h(x) \xrightarrow{3'} h(b)$, the pCTRS is not terminating anymore, but it is still locally confluent and not prefix-preserving.

As seen above, the prefix-constraints of a pCTRS could be annoying to get local confluence. However, they could also be favorable.

▶ **Example 12.** The TRS $R = \{f(a) \to c, a \to b\}$ is not locally confluent because $f(a) \to_R c$, $f(a) \to_R f(b)$, and c and $f(b)$ are irreducible. Actually there is a critical pair $(c, f(b))$, which is not convergent.

Now, Consider the pCTRS $R' = \{\{\epsilon\} : f(a) \to c, \{\epsilon\} : a \to b\}$. Now there is only one derivation issued from $f(a)$, i.e. $f(a) \hookrightarrow_{R'} c$, because the occurrence of a in $f(a)$ is forbidden for the second rule. Actually, the pCTRS R' is locally confluent, and the previous critical pair $(c, f(b))$ is not relevant for R'.

The definition of critical pairs should be modified to fit pCTRSs.

▶ **Definition 13.** (critical pair for a pCTRS) Let $L_1 : l_1 \to r_1$ and $L_2 : l_2 \to r_2$ be prefix-constrained rewrite rules such that $l_1|_p$ and l_2 are unifiable for $\forall p \in PosNonVar(l_1)$. Let $\sigma = mgu(l_1|_p, l_2)$ and $L = \{u \in L_1 \mid u.path(l_1, p) \in L_2\}$. If $L \neq \emptyset$, the triple $(\sigma(r_1), \sigma(l_1)[\sigma(r_2)]_p, L)$ is called a *critical pair*.

Let us notice that L is necessarily regular.

When considering the rules of the pCTRS R' of Example 12, we get $p = 1$ and $L = \emptyset$. Therefore the critical pair $(c, f(b))$ of the TRS R does not produce a critical pair for the pCTRS R'.

If there is a peak coming from an overlap at a non-variable position, then there is a critical pair.

▶ **Lemma 14** (extended critical pair lemma). *Let $R = \{L_1 : l_1 \to r_1, L_2 : l_2 \to r_2\} \cup R'$ be a pCTRS. If $t \hookrightarrow_{[p_1, L_1: l_1 \to r_1, \sigma_1]} t_1$ and $t \hookrightarrow_{[p_2, L_2: l_2 \to r_2, \sigma_2]} t_2$ and $p_2 = p_1.v$ with $v \in PosNonVar(l_1)$, then there exists a critical pair (s_1, s_2, L) and a substitution γ such that $path(t, p_1) \in L$ and $t_1 = t[\gamma(s_1)]_{p_1}$ and $t_2 = t[\gamma(s_2)]_{p_1}$.*

Proof. Let us assume $Var(l_1) \cap Var(l_2) = \emptyset$. Since $t \hookrightarrow t_1$, we have $path(t, p_1) \in L_1$ and $t|_{p_1} = \sigma_1(l_1)$. Since $t \hookrightarrow t_2$, we have $path(t, p_2) \in L_2$ and $t|_{p_2} = \sigma_2(l_2)$. Then $\sigma_2(l_2) = t|_{p_2} = (t|_{p_1})|_v = (\sigma_1(l_1))|v = (\sigma_1(l_1|_v))$ because $v \in PosNonVar(l_1)$. Let us write $\theta = \sigma_1 \cup \sigma_2$. Then $l_1|_v$ and l_2 are unifiable by θ and there exists a substitution γ s.t $\theta = \gamma \circ mgu(l|_v, l_2)$. Let us write $L = \{u \in L_1 \mid u.path(l_1, v) \in L_2\}$ and $\alpha = mgu(l_1|_v, l_2)$. Then $path(t, p_1) \in L$ because $path(t, p_1) \in L_1$ and $path(t, p_2) = path(t, p_1).path(l_1, v) \in L_2$. Then $L \neq \emptyset$, consequently $(\alpha(r_1), (\alpha(l_1)[\alpha(r_2)]_v, L)$ is a critical pair. Let us write $s_1 = \alpha(r_1)$ and $s_2 = \alpha(l_1)[\alpha(r_2)]_v$. Then $t[\gamma(s_1)]_{p_1} = t[\gamma(\alpha(r_1))]_{p_1} = t[\theta(r_1)]_v = t[\sigma_1(r_1)]_v = t_1$. Moreover $t[\gamma(s_2)]_{p_1} = t[\gamma(\alpha(l_1))[\gamma(\alpha(r_2))]_v]_{p_1} = t[\theta(l_1)[\theta(r_2)]_v]_{p_1} = t[\sigma_1(l_1)[\sigma_2(r_2)]_v]_{p_1} = t[\sigma_2(r_2)]_{p_1.v} = t[\sigma_2(r_2)]_{p_2} = t_2$ ◄

Conversely, if there is a critical pair, then there is a peak.

▶ **Lemma 15.** *Let $L_1 : l_1 \to r_1$ and $L_2 : l_2 \to r_2$ be prefix-constrained rewrite rules. If $(\sigma(r_1), \sigma(l_1)[\sigma(r_2)]_v, L)$ is a critical pair, then for each term t and $p_1 \in Pos(t)$:*

$$t|_{p_1} = \sigma(l_1) \wedge path(t, p_1) \in L \implies t \hookrightarrow_{[p_1, L_1:l_1 \to r_1]} t[\sigma(r_1)]_{p_1} \wedge t \hookrightarrow_{[p_1.v, L_2:l_2 \to r_2]} t[\sigma(r_2)]_{p_1.v}$$

Note that at least one pair (t, p_1) exists since $L \neq \emptyset$.

Proof. Since $L \subseteq L_1$ then $path(t, p_1) \in L_1$ therefore $t \hookrightarrow_{[p_1, L_1:l_1 \to r_1]} t[\sigma(r_1)]_{p_1}$. On the other hand, $path(t, p_1.v) = path(t, p_1).path(t|_{p_1}, v) = path(v, p_1).path(l_1, v)$. Moreover, $path(t, p_1) \in L$, consequently $path(t, p_1.v) \in L_2$. $t|_{p_1.v} = (t|_{p_1})|_v = (\sigma(l_1))|_v = \sigma(l_1|_v) = \sigma(l_2)$. Therefore $t \hookrightarrow_{[p_1.v, L_2:l_2 \to r_2]} t[\sigma(r_2)]_{p_1.v}$ ◄

▶ **Definition 16.** *The critical pair (s_1, s_2, L) is said convergent if*

$$\forall t \in T(\Sigma, X), \forall p \in Pos(t), (path(t, p) \in L \implies t[s_1]_p \downarrow_R t[s_2]_p)$$

▶ **Theorem 17.** *(extended Knuth-Bendix's theorem) Let R be a prefix-preserving pCTRS. R is locally confluent if and only if all critical pairs of R are convergent.*

Proof.

1. "$\implies$"
   Let us write $s_1 = \sigma(r_1)$ and $s_2 = \sigma(l_1[\sigma(r_2)]_v$ and $t' = t[\sigma(l_1)]_p$. Through Lemma 15 applied on t' and p, we get $t' \hookrightarrow t'[s_1]_p = t[s_1]_p$ and $t' \hookrightarrow t'[\sigma(r_2)]_{p.v} = t[\sigma(l_1)[\sigma(r_2)]_v]_p = t[s_2]_p$. Through the local confluence property, $t[s_1]_p \downarrow_R t[s_2]_p$.
2. "$\impliedby$"
   Assume $t \hookrightarrow_{[p_1, L1:l_1 \to r_1, \sigma_1]} t_1$ and $t \hookrightarrow_{[p_2, L2:l_2 \to r_2, \sigma_2]} t_2$
   - if $p_1 || p_2$, through Lemma 6, $t_1 \hookrightarrow t_3 \hookleftarrow t_2$
   - without loss of generality, assume $p_1 < p_2$
     - if $p_2.p_1 \notin PosNonVar(l_1)$, through Lemma 10, we have $t_1 \hookrightarrow^* t_4 \hookleftarrow^* t_2$.
     - otherwise through lemma 14, there exist a substitution γ, and a critical pair (s_1, s_2, L), s.t. $path(t, p_1) \in L$ and $t_1 = t[\gamma(s_1)]_{p_1}$ and $t_2 = [\gamma(s_2)]_{p_1}$. This critical pair is convergent and since $path(t, p_1) \in L$, we have $t[s_1]_{p_1} \hookrightarrow^* \hookleftarrow^* t[s_2]_{p_1}$. Since, pCTRS rewriting is stable through instantiation, $t_1 \hookrightarrow^* \hookleftarrow^* t_2$ ◄

In general, to check the convergence of a critical pair according to Definition 16, infinitely many contexts t should be tried, which is impossible. Therefore we need to define a stronger sufficient condition. Let us first introduce the notion of rewriting under a prefix language. As usual, for a string w and a string language L, we define $L.w$ by $L.w = \{v.w \mid v \in L\}$.

▶ **Definition 18.** Let $R = \{L : l \to r\} \cup R'$ be a pCTRS, and $L' \subseteq Dir(\Sigma)^*$.
$t \xrightarrow{L'}_{[p, L:l \to r, \sigma]} t'$ if $L'.path(t, p) \subseteq L$ and σ is a substitution s.t. $t|_p = \sigma(l)$ and $t' = t[\sigma(r)]_p$.
We also write $t \xrightarrow{L'}_R t'$, and $\xrightarrow{L'}{}^*_R$ will denote the reflexive-transitive closure of $\xrightarrow{L'}_R$.

▶ **Remark.** $t \hookrightarrow_{[p, L:l \to r]} t' \iff t \xrightarrow{\{\epsilon\}}_{[p, L:l \to r]} t'$.

▶ **Lemma 19.** *If $t \xrightarrow{L'}_{[p, L:l \to r]} t'$ then*

$$\forall t_0 \in T(\Sigma, X), \forall p' \in Pos(t_0), (path(t_0, p') \in L' \implies t_0[t]_{p'} \hookrightarrow_{[p'.p, L:l \to r]} t_0[t']_{p'})$$

Proof. Through the hypothesis $t \xrightarrow{L'}_{[p, L:l \to r]} t'$, we have $t \to_{[p, l \to r]} t'$ and $L'.path(t, p) \subseteq L$.
Assume $path(t_0, p') \in L'$. Then $path(t_0[t]_{p'}, p'.p) = path(t_0, p').path(t, p) \in L$. Then
$t_0[t]_{p'} \hookrightarrow_{[p'.p, L:l \to r]} t_0[t']_{p'}$. ◀

▶ **Corollary 20.** *If $L' \neq \emptyset$ and $t_1 \xrightarrow{L'}_R t_2 \xrightarrow{L'}_R \cdots \xrightarrow{L'}_R t_n$,*
then there exists $t_0 \in T(\Sigma, X)$ and $p' \in Pos(t_0)$ s.t. $t_0[t_1]_{p'} \hookrightarrow_R t_0[t_2]_{p'} \hookrightarrow_R \cdots \hookrightarrow_R t_0[t_n]_{p'}$.

▶ **Corollary 21.** *If $L' \neq \emptyset$ and $\xrightarrow{L'}_R$ is not terminating, then $\hookrightarrow_R$ is not terminating.*
Consequently, if $\hookrightarrow_R$ is terminating, then $\xrightarrow{L'}_R$ is terminating.

▶ **Definition 22.** The critical pair (s_1, s_2, L) is said *strongly convergent* if there exits a term
t such that $s_1 \xrightarrow{L}{}^*_R t$ and $s_2 \xrightarrow{L}{}^*_R t$.

Therefore, if the pCTRS R is terminating, the strong convergence of a critical pair (s_1, s_2, L)
can be checked by computing all descendants of s_1 and of s_2 under prefix-language L, which
are finitely many, and looking for common elements.

▶ **Lemma 23.** *Strong convergence implies convergence.*

Proof. We have $s_1 \xrightarrow{L}{}^* s_3$ and $s_2 \xrightarrow{L}{}^* s_3$. Let $t_0 \in T(\Sigma), p \in Pos(t_0)$ s.t $path(t_0, p) \in L$.
Through corollary 20
$t_0[s_1]_p \hookrightarrow^* t_0[s_3]_p$ and $t_0[s_2]_p \hookrightarrow^* t_0[s_3]_p$ then the critical pair is convergent. ◀

▶ **Theorem 24.** *Let R be a prefix-preserving pCTRS.*
If all critical pairs of R are strongly convergent, then R is locally confluent.

Proof. It is naturally deduced from Lemma 23 and Theorem 17. ◀

Let us note that the converse is wrong as illustrated by the following Example .

▶ **Example 25.** Consider the pCTRS

$$R = \{\{\langle h, 1 \rangle\} : f(a) \to c, \{\langle h, 1 \rangle.\langle f, 1 \rangle\} : a \to b, \{\epsilon\} : h(f(b)) \to h(c)\}$$

R is prefix-preserving since the rewrite rules do not contain variables.
There is only one critical pair $(c, f(b), \{\langle h, 1 \rangle\})$, which is convergent because $h(f(b)) \hookrightarrow_R h(c)$.
From Theorem 17, R is locally confluent. However, the critical pair is not strongly convergent
since according to Definition 18, c and $f(b)$ are irreducible by $\xrightarrow{\{\langle h, 1 \rangle\}}_R$.

The context-sensitive case

In this section we compare the previous results with those of [11]. A CS-TRS (R_0, μ) may be viewed as a particular pCTRS $R = \{L_k : l_k \to r_k \mid (l_k \to r_k) \in R_0\}$, where all languages L_k are the same (say L), and L is composed of all strings (including the empty string) over the alphabet $\{\langle f, i \rangle \mid f \in \Sigma, i \in \mu(f)\}$.

▶ **Example 26.**

$R_0 = \{f(h(x,y),y) \to g(x,y), h(a,b) \to i(a), a \to b\} s.t.$

$\mu(f) = \mu(h) = 1,$

then we can view this CS-TRS as the following pCTRS

$R = \{L : f(h(x,y),y) \to g(x,y), L : h(a,a) \to i(a), L : a \to b\} s.t.$

$L = (\langle f, 1 \rangle \mid \langle h, 1 \rangle)^*.$

In this framework, note that $u, v, w \in L \iff u.v.w \in L$, and $path(t, p) \in L \iff p \in Pos^\mu(t)$. Consequently, the pCTRS R is prefix-preserving if and only if the CS-TRS (R_0, μ) is with left homogeneous replacing variables (Definition 4.4 of [11]).

According to Definition 13, a critical pair between $L : l_1 \to r_1$ and $L : l_2 \to r_2$ is of the form $(\sigma(r_1), \sigma(l_1)[\sigma(r_2)]_p, L')$ where $L' = \{u \in L \mid u.path(l_1, p) \in L\} \neq \emptyset$. Since $L' \neq \emptyset$, there exists at least one string $u \in L$ s.t. $u.path(l_1, p) \in L$. Therefore $path(l_1, p) \in L$, then $p \in Pos^\mu(l_1)$. On the other hand, for all $u \in L$, we have $u.path(l_1, p) \in L$ (because $path(l_1, p) \in L$). Consequently $L' = L$.

For instance, for the example 26 we have two critical pairs

$(f(i(a), a)), g(a, a), L)$
$(h(b, a), i(a), L).$

Since all critical pairs have the same language, we can omit it, and we get the same notion (called μ-critical pair) as in Definition 4.7 of [11].

Therefore, Theorem 17 gives the same result as Theorem 4.9 of [11]. However, as mentioned previously, with a pCTRS infinitely many contexts should be tried to check the convergence of a critical pair. Fortunately $\epsilon \in L$, and when using Definition 16 we can consider p as the root position, i.e. the critical pair should also converge without context. Conversely, if the critical pair converges without context, it also converges with any context t assuming $p \in Pos^\mu(t)$, which holds because $path(t, p) \in L$ is assumed. Thus, the convergence of a critical pair can be checked without using a context, i.e. as in [11].

As a conclusion, if the pCTRS is context-sensitive, thanks to Theorem 17 we get the same result as [11]. If the pCTRS is not context-sensitive, we can ensure local-confluence using Theorem 24.

4 Working with String Automata

From an operational point of view, Section 3 does not say anything for handling prefix languages, for checking whether a pCTRS is prefix-preserving, for computing the language of a critical pair, and for computing $\overset{L}{\hookrightarrow}_R$ steps. This is why we consider in this section that for a pCTRS $R = \{L_k : l_k \to r_k \mid 1 \leq k \leq n\}$, each language $L_k \subseteq Dir(\Sigma)^*$ is defined by a finite string automaton $\mathcal{A}^k = (Dir(\Sigma), Q^k, Q^k_I, Q^k_f, \Delta^k)$.

For efficiency of further computations, we assume that each each $\mathcal{A}^k$ is deterministic and complete. In other words, we consider that the automata are determinized and completed at the beginning, and the new automata generated by the computations will be still deterministic and complete.

For any automaton $\mathcal{A} = (\Sigma, Q, Q_I, Q_f, \Delta)$ and $q \in Q$, let us define $L_{\mathcal{A}}(q) = \{w \in \Sigma^* \mid \exists q_f \in Q_f, (q, w) \mapsto^*_\Delta (q_f, \epsilon)\}$. Note that $L_{\mathcal{A}} = \cup_{q_I \in Q_I} L_{\mathcal{A}}(q_I)$, and $L_{\mathcal{A}}(q)$ is recognized by the automaton $(\Sigma, Q, \{q\}, Q_f, \Delta)$.

Let w be a string and $L \subseteq \Sigma^*$ be a string language. Let us define $L^{w-} = \{u \in \Sigma^* \mid w.u \in L\}$ and $L^{-w} = \{u \in \Sigma^* \mid u.w \in L\}$.

For a string w, let $Q_I^{w-} = \{q \in Q \mid \exists q_I \in Q_I, (q_I, w) \mapsto^*_\Delta (q, \epsilon)\}$, and let $Q_f^{-w} = \{q \in Q \mid \exists q_f \in Q_f, (q, w) \mapsto^*_\Delta (q_f, \epsilon)\}$. Let $\mathcal{A}^{w-} = (\Sigma, Q, Q_I^{w-}, Q_f, \Delta)$ and $\mathcal{A}^{-w} = (\Sigma, Q, Q_I, Q_f^{-w}, \Delta)$.

▶ **Lemma 27.** $L_{\mathcal{A}^{w-}} = (L_{\mathcal{A}})^{w-}$ and $L_{\mathcal{A}^{-w}} = (L_{\mathcal{A}})^{-w}$. Moreover, if $\mathcal{A}$ is deterministic and complete, so are $\mathcal{A}^{w-}$ and $\mathcal{A}^{-w}$.

Proof.
1. Let us prove that $L_{\mathcal{A}^{w-}} \subseteq (L_{\mathcal{A}})^{w-}$
   Let $u \in L_{\mathcal{A}^{w-}}$. There exists $q \in Q_I^{w-}$ and $q_f \in Q_f$ such that $(q, u) \mapsto^*_\Delta (q_f, \epsilon)$. Then there exists $q_I \in Q_I$ such that $(q_I, w) \mapsto^*_\Delta (q, \epsilon)$. Consequently, $(q_I, w.u) \mapsto^*_\Delta (q, u) \mapsto^*_\Delta (q_f, \epsilon)$. Then $w.u \in L_{\mathcal{A}}$, then $u \in (L_{\mathcal{A}})^{w-}$.
2. Let us prove $(L_{\mathcal{A}})^{w-} \subseteq L_{\mathcal{A}^{w-}}$
   Let $u \in (L_{\mathcal{A}})^{w-}$. Then $w.u \in L_{\mathcal{A}}$. Consequently, there exists $q_I \in Q_I$, $q_f \in Q_f$ and $q \in Q$ such that $(q_I, w.u) \mapsto^*_\Delta (q, u) \mapsto^*_\Delta (q_f, \epsilon)$. Then $(q_I, w) \mapsto^*_\Delta (q, \epsilon)$, that is $q \in Q_I^{w-}$. Consequently, $u \in (L_{\mathcal{A}})^{w-}$.
3. If $\mathcal{A}$ is deterministic and complete, then $|Q_I^{w-}| = |\{q \in Q \mid \exists q_I \in Q_I, (q_I, w) \mapsto^*_\Delta (q, \epsilon)\}|$, where $|A|$ denotes the number of elements of the set A, as usual. But $|Q_I| = 1$ and Δ is the set of the transitions in $\mathcal{A}$. Then $|Q_I^{w-}| = 1$. On the other part, the transitions in $\mathcal{A}^{w-}$ and $\mathcal{A}$ are the same.
4. Let us prove $L_{\mathcal{A}^{-w}} \subseteq (L_{\mathcal{A}})^{-w}$
   Let $u \in L_{\mathcal{A}^{-w}}$. There exists $q_I \in Q_I$, $q \in Q_f^{-w}$ such that $(q_I, u) \mapsto^*_\Delta (q, \epsilon)$. Then there exists $q_f \in Q_f$, $(q, w) \mapsto^*_\Delta (q_f, \epsilon)$. Consequently, $(q_I, u.w) \mapsto^*_\Delta (q, w) \mapsto^*_\Delta (q_f, \epsilon)$. Then $u.w \in L_{\mathcal{A}}$, the $u \in (L_{\mathcal{A}})^{-w}$.
5. $(L_{\mathcal{A}})^{-w} \subseteq L_{\mathcal{A}^{-w}}$
   Let $u \in (L_{\mathcal{A}})^{-w}$. Then $u.w \in L_{\mathcal{A}}$. Consequently, there exists $q_I \in Q_I$, $q_f \in Q_f$ and $q \in Q$ such that $(q_I, u.w) \mapsto^*_\Delta (q, w) \mapsto^*_\Delta (q_f, \epsilon)$. Then $q \in Q_f^{-w}$ and $(q_I, u) \mapsto^*_\Delta (q, \epsilon)$. Consequently, $u \in L_{\mathcal{A}^{-w}}$.
6. The initial states and the transitions of $\mathcal{A}^{-w}$ and $\mathcal{A}$ are the same.   ◀

Prefix preserving.   To check whether a pCTRS is prefix-preserving (Definition 8), we use the following result.

▶ **Theorem 28.** *For the prefix-constrained rewrite rules $L_1 : l_1 \to r_1$ and $L_2 : l_2 \to r_2$, let $\mathcal{A}_1 = (\Sigma, Q^1, Q_I^1, Q_f^1, \Delta^1)$ and $\mathcal{A}_2 = (\Sigma, Q^2, Q_I^2, Q_f^2, \Delta^2)$ be deterministic and complete automata that recognize L_1 and L_2 respectively. Let $S_{1,2} = \{q \in Q^2 \mid \exists (q_I^1, q_I^2) \in Q_I^1 \times Q_I^2, \exists u \in Dir(\Sigma)^*, \exists q_f^1 \in Q_f^1, ((q_I^1, q_I^2), u) \mapsto^*_{\Delta^1 \otimes \Delta^2} ((q_f^1, q), \epsilon)\}$, which can be computed by saturating $\{(q_I^1, q_I^2)\}$ with the transitions of $\Delta^1 \otimes \Delta^2$.*

The pCTRS R is prefix-preserving if and only if for all rewrite rules $L_1 : l_1 \to r_1$ and $L_2 : l_2 \to r_2$ of R, $\forall x \in Var(l_1)$, $\forall p, p' \in Pos(l_1, x)$, $\forall p'' \in Pos(r_1, x)$, $\forall q \in S_{1,2}$:

$$L_{\mathcal{A}_2}(q)^{path(l_1, p)-} = L_{\mathcal{A}_2}(q)^{path(l_1, p')-} \subseteq L_{\mathcal{A}_2}(q)^{path(r_1, p'')-}$$

Proof.

- Let us begin by proving the implication "$\Longrightarrow$".
  Let us assume $u \in L_1$ and $u.path(l_1, p).w \in L_2$. Then there exists $q_I^1 \in Q_I^1$, $q_I^2 \in Q_I^2, q_f^1 \in Q_f^1, q \in Q^2$ such that $(q_I^1, u) \to^*_{\Delta^1} (q_f^1, \epsilon)$ and $(q_I^2, u) \to^*_{\Delta^2} (q, \epsilon)$. Then $q \in S_{1,2}$. Moreover, $path(l_1, p).w \in L_{\mathcal{A}_2}(q)$, then $w \in L_{\mathcal{A}_2}(q)^{path(l_1,p)-}$. Through the hypothesis, we have $w \in L_{\mathcal{A}_2}(q)^{path(l_1,p')-}$ and $w \in L_{\mathcal{A}_2}(q)^{path(l_1,p'')-}$. Then $path(l_1, p').w \in L_{\mathcal{A}_2}(q)$ and $path(r_1, p'').w \in L_{\mathcal{A}_2}(q)$. As $(q_I^2, u) \to^*_{\Delta^2} (q, \epsilon)$, we get $u.path(l_1, p').w \in L_{\mathcal{A}_2} = L_2$ and $u.path(r_1, p'').w \in L_{\mathcal{A}_2} = L_2$. Consequently, R is prefix-preserving.
- Let us continue by proving the implication "$\Longleftarrow$".
  Let $q \in S_{1,2}$ and $w \in L_{\mathcal{A}_2}(q)^{path(l_1,p)-}$. There exists $u \in Dir(\Sigma)^*$, $q_I^1 \in Q_I^1$, $q_I^2 \in Q_I^2, q_f^1 \in Q_f^1$ such that $(q_I^1, u) \to^*_{\Delta^1} (q_f^1, \epsilon)$ and $(q_I^2, u) \to^*_{\Delta^2} (q, \epsilon)$. Then $u \in L_1$. Moreover $path(l_1, p).w \in L_{\mathcal{A}_2}(q)$, then there exists $q_f^2 \in Q_f^2$ such that $(q, path(l_1, p).w) \to^*_{\Delta^2} (q_f^2, \epsilon)$. Consequently, $u.path(l_1, p).w \in L_2$. As R is prefix-preserving, we have $u.path(l_1, p').w \in L_2 = L_{\mathcal{A}_2}$ and $u.path(r_1, p'').w \in L_2 = L_{\mathcal{A}_2}$. Then $path(l_1, p').w \in L_{\mathcal{A}_2}(q)$ and $path(r_1, p'').w \in L_{\mathcal{A}_2}(q)$, then $w \in L_{\mathcal{A}_2}(q)^{path(l_1,p')-}$ and $w \in L_{\mathcal{A}_2}(q)^{path(r_1,p'')-}$. Therefore, $L_{\mathcal{A}_2}(q)^{path(l_1,p)-} \subseteq L_{\mathcal{A}_2}(q)^{path(l_1,p')-}$ and $L_{\mathcal{A}_2}(q)^{path(l_1,p)-} \subseteq L_{\mathcal{A}_2}(q)^{path(r_1,p'')-}$. On the other hand, we prove that $L_{\mathcal{A}_2}(q)^{path(l_1,p')-} \subseteq L_{\mathcal{A}_2}(q)^{path(l_1,p)-}$ by exchanging p and p'.   ◀

Critical pair.   Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two deterministic and complete automata that recognize the languages L_1 and L_2 of the two rules in the critical pair Definition 13. Then the language L of the critical pair is recognized by the automaton $\mathcal{A}_1 \cap \mathcal{A}_2^{-path(l_1,p)}$, which is still deterministic and complete.

Rewriting under context.   Let $\mathcal{A}'$ and $\mathcal{A}$ be deterministic and complete automata that recognizes the languages L' and L of Definition 18. To check whether $L'.path(t, p) \subseteq L$, we use the following result:

▶ **Lemma 29.** $L'.path(t, p) \subseteq L \iff L_{(\mathcal{A}' \cap (\bar{\mathcal{A}})^{-path(t,p)})} = \emptyset$.

Proof.

1. "$\Longleftarrow$"
   By contradiction. Let us assume there exists $u \in L'$ such that $u.path(t, p) \notin L$. Then $u.path(t, p) \in \bar{L} = L_{\bar{\mathcal{A}}}$. Then $u \in (L_{\bar{\mathcal{A}}})^{-path(t,p)} = L_{(\bar{\mathcal{A}})^{-path(t,p)}}$. Since $u \in L'$, we have $u \in L_{\mathcal{A}'}$, then $u \in L_{\mathcal{A}' \cap (\bar{\mathcal{A}})^{-path(t,p)}} = \emptyset$ according to the hypothesis. Contradiction.
2. "$\Longrightarrow$"
   By contradiction. Let assume there exists $u \in L_{\mathcal{A}' \cap (\bar{\mathcal{A}})^{-path(t,p)}}$. Then $u \in L'$ and $u.path(t, p) \in \bar{L}$, that is $u.path(t, p) \notin L$ and $u.path(t, p) \in L'.path(t, p)$. Consequently, $L'.path(t, p) \not\subseteq L$. Contradiction.   ◀

Note that checking equalities or inclusions of languages as in Theorem 28, or computing the complement as in Lemma 29, is polynomial since automata are deterministic and complete.

5   Extended Knuth-Bendix Completion

The goal of extended completion is to transform an arbitrary initial pCTRS R (or a set of equalities) into a confluent and terminating pCTRS R' without changing the equality modulo the pCTRS, i.e. such that $=_R$ and $=_{R'}$ are identical. To do it, we use the result of Theorem 24, therefore the pCTRS needs to be prefix-preserving. However, with the notations

of Definition 8, whenever $u \in L_1 \wedge u.path(l_1, p).w \in L_2$ whereas $u.path(r_1, p'').w \notin L_2$, i.e. the pCTRS is not prefix-preserving, we could make it prefix-preserving by extending L_2 into L_2' so that $u.path(r_1, p'').w \in L_2'$. Unfortunately, this may change the equality modulo the pCTRS.

▶ **Example 30.** Let $\Sigma = \{f, g, a, b, c, d\}$ and

$$R = \{\{\epsilon\} : f(x) \overset{1}{\to} g(x, c), \ \{\langle f, 1 \rangle\} : a \overset{2}{\to} b\}$$

Let L_1 and L_2 be the prefix-languages of rules 1 and 2 respectively. R is not prefix-preserving because, let $u = \epsilon \in L_1$ and $w = \epsilon$, and we have $u.path(f(x), 1).w = \langle f, 1 \rangle \in L_2$ whereas $u.path(g(x, c), 1).w = \langle g, 1 \rangle \notin L_2$. Note that R is not confluent since there is the peak $f(a) \hookrightarrow g(a, c)$ by rule 1, and $f(a) \hookrightarrow f(b) \hookrightarrow g(b, c)$ by rules 1 and 2, which is not convergent since $g(a, c)$ and $g(b, c)$ are irreducible.

Now let us extend L_2 by considering the pCTRS:

$$R' = \{\{\epsilon\} : f(x) \overset{1}{\to} g(x, c), \ \{\langle f, 1 \rangle\} \cup \{\langle g, 1 \rangle\} : a \overset{2'}{\to} b\}$$

R' is prefix-preserving. However $g(a, d) \hookrightarrow_{R'} g(b, d)$ whereas $g(a, d) \neq_R g(b, d)$. In other words, $=_R$ and $=_{R'}$ are not identical.

This difficulty may depend on the orientation of rewrite rules. The pCTRS R of Example 30 is terminating, however let us reverse the first rule of R, i.e. let

$$R'' = \{\{\epsilon\} : g(x, c) \overset{1''}{\to} f(x), \ \{\langle f, 1 \rangle\} : a \overset{2}{\to} b\}$$

R'' is prefix-preserving and is also terminating. Moreover $=_R$ and $=_{R''}$ are identical. Unfortunately, changing the orientation does not always make the pCTRS prefix-preserving, and may not preserve termination. In other words, extended completion will fail when one cannot get a prefix-preserving pCTRS.

The usual Knuth-Bendix completion generates an inter-reduced TRS R, which means (roughly) that the left-hand-side and the right-hand-side of each rule of R are not reducible by the other rules of R. This notion cannot be extended to pCTRSs in an easy way.

▶ **Example 31.** Consider the TRS $R = \{f(x) \overset{1}{\to} g(x), g(x) \overset{2}{\to} h(x)\}$. Then R is not inter-reduced since the right-hand-side of rule 1 is reducible by rule 2. Now let

$$R' = \{\{\langle i, 1 \rangle\} \cup \{\langle j, 1 \rangle\} : f(x) \overset{1'}{\to} g(x), \ \{\langle i, 1 \rangle\} : g(x) \overset{2'}{\to} h(x)\}$$

So, the right-hand-side of rule 1 is reducible by rule 2 under context i, but not under context j. An inter-reduced pCTRS R'' such that $=_{R'}$ and $=_{R''}$ are identical, could be

$$R'' = \{\{\langle i, 1 \rangle\} : f(x) \overset{1''}{\to} h(x), \ \{\langle j, 1 \rangle\} : f(x) \overset{1'''}{\to} g(x), \ \{\langle i, 1 \rangle\} : g(x) \overset{2''}{\to} h(x)\}$$

In this paper, we present a basic extended Knuth-Bendix completion, which does not attempt to produce an inter-reduced pCTRS. It is described by inference rules, as in [2], and computes (P_{i+1}, R_{i+1}) from (P_i, R_i) using a derivation relation denoted $\vdash$, where P_i, P_{i+1} are sets of prefix-constrained equalities of the form $L : p = q$[4] and R_i, R_{i+1} are sets of prefix-constrained rewrite rules of the form $L : l \to r$.

---

[4] We assume that $=$ is commutative, i.e. $L : p = q$ is the same equality as $L : q = p$.

1. **Orient**

$$\frac{P \cup \{L : p = q\},\ R}{P,\ R \cup \{L : p \to q\}} \quad \text{if } R \cup \{L : p \to q\} \text{ is prefix-preserving and terminating}$$

2. **Deduce**

$$\frac{P,\ R}{P \cup \{L : p = q\},\ R} \quad \text{if } (p, q, L) \text{ is a critical pair between rules of } R$$

3. **Simplify**

$$\frac{P \cup \{L : p = q\},\ R}{P \cup \{L : p' = q\},\ R} \quad \text{if } p \xrightarrow{L}_R p'$$

4. **Delete**

$$\frac{P \cup \{L : p = p\},\ R}{P,\ R}$$

Orient needs to check that $R \cup \{L : p \to q\}$ is terminating. This can be done by transforming the pCTRS into an ordinary TRS [1], which preserves termination, and checking the termination of the ordinary TRS using the usual techniques and tools. This transformation can even be done incrementally: each time *Orient* is run, new rewrite rules are added into the ordinary TRS.

With our basic completion above, inference rules *Simplify* and *Delete* are only applied on (non-oriented) equations of P. During the completion procedure, oriented rules of R are neither simplified nor deleted.

▶ **Lemma 32** (Soundness). *If $(P, R) \vdash (P', R')$, then $=_{P \cup R}$ and $=_{P' \cup R'}$ are identical.*

Proof.
1. *Orient* :
   $t =_{L:p=q} t' \iff t =_{L:p \to q} t'$ because $=_{L:p=q}$ is a symmetric relation.
2. *Deduce* :
   Let us consider the critical pair (p, q, L) obtained from the rewriting rules $L_1 : l_1 \to r_1 \in R$ and $L_2 : l_2 \to r_2 \in R$. Then $(p, q, L) = (\sigma(r_1), \sigma(l_1)[\sigma(r_2)]_v, L)$. If $t' =_{[p', L:p=q, \theta]} t''$, then $t' = t_0[\theta(p)]_{p'}, t'' = t_0[\theta q]_{p'}$ and $path(t_0, p') \in L$. Let us write $t = t_0[\sigma(l_1)]_{p'}$. Through Lemma 15, we have $t \hookrightarrow_{[p', L_1:l_1 \to r_1]} t[\sigma(r_1)]_{p'} = t_0[p]_{p'}$ and $t \hookrightarrow_{[p'.v, L_2:l_2 \to r_2]} t[\sigma(r_2)]_{p'.v} = t_0[\sigma(l_1[\sigma(r_2)]_v)_{p'}) = t_0[q]_{p'}$. Since $t' =_{[p', L:p=q, \theta]} t''$, let us assume $Var(p) \cap Var(t') = \emptyset$ and $Var(q) \cap Var(t'') = \emptyset$. Then $Var(p) \cap Var(t_0) = \emptyset$ and $Var(q) \cap Var(t_0) = \emptyset$. Consequently, $\theta(t) \hookrightarrow_{[p', L_1:l_1 \to r_1]} t_0[\theta(p)]_{p'} = t'$ and $\theta(t) \hookrightarrow_{[p'.v, L_2:l_2 \to r_2]} t_0[\theta(q)]_{p'} = t''$. Then $t' =_R t''$.
3. *Simplify*
   a. " $\implies$ "
      If $t =_{[u, L:p=q, \sigma]} t'$, then we have $t|_u = \sigma(p)]$ and $t' = t[\sigma(q)]$ and $path(t, u) \in L$. But $p \xrightarrow{L}_{[v, L':l' \to r', \theta]} p'$, then $p|_v = \theta(l')$ and $p' = p[\theta(r')]_v$ and $L.path(p, v) \subseteq L'$. Consequently, $t \hookrightarrow_{[u.v, L':l' \to r', \sigma\theta]} t[\sigma(\theta(p'))]_{u.v} = t[t|_u[\sigma(\theta(r'))]_v]_u = t[(\sigma(p)[\sigma(\theta(r'))]_v]_u = t[\sigma(p([\theta(r'))]_v]_u = t[\sigma(p')]_u$ since $path(t, u.v) = path(t, u).path(p, v) \in L'$ (note that $path(t, u) \in L$). Furthermore, $t[\sigma(p')]_u =_{[L, p'=q]} t[\sigma(q)]_u = t'$ since $path(t, u) \in L$. Consequently $t =_{R \cup \{L:p'=q\}} t'$.
   b. " $\impliedby$ "
      The converse is similar since the direction of the rewrite step $p \hookrightarrow p'$ does not matter.
4. *Delete*
   If $t =_{L:p=p} t'$, then $t = t'$. Thus $t =_{P \cup R} t'$.   ◀

Consider a derivation $(P_0, R_0) \vdash \cdots \vdash (P_n, R_n)$. Note that $R_0 \subseteq R_1 \subseteq \cdots \subseteq R_n$.

▶ **Lemma 33.** *(Completeness) Let (p, q, L) be a critical pair between some rules of R_n. If $p \overset{L}{\hookrightarrow}_{R_n}^* p'$ and $q \overset{L}{\hookrightarrow}_{R_n}^* q'$, then (p, q, L) is strongly convergent in $R_n \cup \{L : p' \to q'\}$.*

Proof. $p' \overset{L}{\hookrightarrow}_{[\epsilon, p' \to q']} q'$ since $L.path(p', \epsilon) = L \subseteq L$. Consequently, $p \overset{L}{\hookrightarrow}_{R_n}^* p' \overset{L}{\hookrightarrow}_{[\epsilon, L : p' \to q']} q'$ and $q \overset{L}{\hookrightarrow}_{R_n}^* q'$. Then the critical pair is strongly convergent in $R_n \cup \{L : p' \to q'\}$. ◀

Fairness hypothesis. $(P_0, R_0) \vdash \cdots \vdash (P_n, R_n)$ is *fair* if for all critical pair (p, q, L) between rules of R_n, there is some $i \in \{0, \dots, n\}$ such that $(L : p = q) \in P_i$. In other words, all critical pairs have been computed thanks to *Deduce*. From Lemmas 32, 33 and Theorems 24, 4 we get:

▶ **Corollary 34.** *If $(P_0, R_0) \vdash \cdots \vdash (P_n, R_n)$ is fair and $R_0 = P_n = \emptyset$, then R_n is confluent and terminating. Moreover the relations $=_{P_0}$ and $=_{R_n}$ are identical.*

However, like the usual Knuth-Bendix completion, the extended Knuth-Bendix completion fails if we cannot obtain $P_n = \emptyset$ for some n. In particular, it arises if *Orient* cannot orient a persistent critical pair because the resulting pCTRS would not be prefix-preserving or would not be terminating.

The above basic completion could be improved by including more inference rules like *Simplifying* and *Deleting* oriented rules of R. However, the proof of correctness and completeness of such completion procedure would be more complicated and could be done, for instance, by extending the proof transformation method of [2].

6 Conclusion and Further Work

In this paper, we present a sufficient condition that ensures the local confluence of prefix-constrained rewrite systems, and consequently the confluence of terminating ones. This result subsumes that of [11] about local-confluence of context-sensitive rewrite systems. Prefix-preserving and critical-pair strong convergence assumptions are sufficient, but are not necessary. Finding weaker assumptions is an interesting challenge.

The second contribution of this paper is an extended Knuth-Bendix completion procedure for prefix-constrained rewrite systems. This procedure could be improved to get inter-reduced systems, by adding some inference rules, which could also improve the efficiency.

Controlled rewriting [7] is an extension of prefix-constrained rewriting, where rewritable positions are defined by a regular tree language that considers the entire term (i.e. not only prefixes). It could be interesting to study local-confluence, and define a completion procedure for controlled rewrite systems.

───── **References** ─────

1 Nirina Andrianarivelo, Vivien Pelletier, and Pierre Réty. Transforming prefix-constrained or controlled rewrite systems. In Mohamed Mosbah and Michaël Rusinowitch, editors, *SCSS 2017, The 8th International Symposium on Symbolic Computation in Software Science 2017, April 6-9, 2017, Gammarth, Tunisia*, volume 45 of *EPiC Series in Computing*, pages 49–62. EasyChair, 2017. URL: http://www.easychair.org/publications/paper/ Transforming_Prefix-constrained_or_Controlled_Rewrite_Systems.

2   Leo Bachmair and Nachum Dershowitz. Completion for rewriting modulo a congruence. In Pierre Lescanne, editor, *Rewriting Techniques and Applications, 2nd International Conference, RTA-87, Bordeaux, France, May 25-27, 1987, Proceedings*, volume 256 of *Lecture Notes in Computer Science*, pages 192–203. Springer, 1987. `doi:10.1007/3-540-17220-3_17`.

3   Alonzo Church and J. B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936. URL: `http://www.jstor.org/stable/1989762`.

4   Kokichi Futatsugi, Joseph A. Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, USA, January 1985*, pages 52–66, 1985. `doi:10.1145/318593.318610`.

5   Jürgen Giesl and Aart Middeldorp. Transformation Techniques for Context-Sensitive Rewrite Systems. *J. Funct. Program.*, 14(4):379–427, 2004. `doi:10.1017/S0956796803004945`.

6   Gérard P. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems: Abstract properties and applications to term rewriting systems. *J. ACM*, 27(4):797–821, 1980. `doi:10.1145/322217.322230`.

7   Florent Jacquemard, Yoshiharu Kojima, and Masahiko Sakai. Controlled Term Rewriting. In Cesare Tinelli and Viorica Sofronie-Stokkermans, editors, *Frontiers of Combining Systems, 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings*, volume 6989 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2011. `doi:10.1007/978-3-642-24364-6_13`.

8   Florent Jacquemard, Yoshiharu Kojima, and Masahiko Sakai. Term Rewriting with Prefix Context Constraints and Bottom-Up Strategies. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *LNCS*, pages 137–151. Springer, 2015. `doi:10.1007/978-3-319-21401-6_9`.

9   D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, pages 342–376. Springer, Berlin, Heidelberg, 1983.

10   S. Lucas. Context-Sensitive Computations in Functional and Functional logic Programs. *Journal of Functional and Logic Programming*, 1998(1), January 1998.

11   Salvador Lucas. Context-sensitive computations in confluent programs. In Herbert Kuchen and S. Doaitse Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs, 8th International Symposium, PLILP'96, Aachen, Germany, September 24-27, 1996, Proceedings*, volume 1140 of *Lecture Notes in Computer Science*, pages 408–422. Springer, 1996. `doi:10.1007/3-540-61756-6_100`.

Fixed-Point Constraints for Nominal Equational Unification

Mauricio Ayala-Rincón[1]
Departments of Mathematics and Computer Science, Universidade de Brasília, Brasília, Brazil

Maribel Fernández
Department of Informatics, King's College London, London, UK

Daniele Nantes-Sobrinho[2]
Departments of Mathematics and Computer Science, Universidade de Brasília, Brasília, Brazil

──── Abstract ────

We propose a new axiomatisation of the alpha-equivalence relation for nominal terms, based on a primitive notion of fixed-point constraint. We show that the standard freshness relation between atoms and terms can be derived from the more primitive notion of permutation fixed-point, and use this result to prove the correctness of the new alpha-equivalence axiomatisation. This gives rise to a new notion of nominal unification, where solutions for unification problems are pairs of a fixed-point context and a substitution. Although it may seem less natural than the standard notion of nominal unifier based on freshness constraints, the notion of unifier based on fixed-point constraints behaves better when equational theories are considered: for example, nominal unification remains finitary in the presence of commutativity, whereas it becomes infinitary when unifiers are expressed using freshness contexts.

1 Introduction

This paper presents a new axiomatisation of α-equivalence for nominal terms via permutation fixed points, and revisits nominal unification in this setting.

In nominal syntax [16], *atoms* are used to represent object-level variables and *atom permutations* to implement renamings, following the nominal-sets approach advocated by Gabbay and Pitts [10, 12, 14]. Atoms can be abstracted over terms, the syntax $[a]s$ represents the abstraction of a in s. To rename an abstracted atom a to b, a *swapping* permutation $\pi = (a\,b)$ is applied. Thus, the action of π over $[a]s$, written as $(a\,b) \cdot [a]s$, produces the nominal term $[b]s'$, where s' is the result of replacing all occurrences of a in s by b, and all occurences of b in s by a. The α-equivalence relation between nominal terms is specified using swappings together with a *freshness relation* between atoms and terms, written $b\#s$, which roughly corresponds to b not occurring free in s.

In this setting, checking α-equivalence requires another first-order specialised calculus to check freshness constraints. For instance, checking whether $[a]s \approx_\alpha [b]t$ reduces to checking

---

whether $s \approx_\alpha (b\,a) \cdot t$ and $a\#t$. The action of a permutation propagates down the structure of nominal terms, until a variable is reached: permutations suspend over variables. Thus, $\pi \cdot s$ represents the action of a permutation over a nominal term, but is not itself a nominal term unless s is a variable; for instance, $\pi \cdot X$ is a *suspension* (also called *moderated variable*), which is a nominal term.

The presence of moderated variables and atom-abstractions makes reasoning about equality of nominal terms more involved than in standard first-order syntax. For example, $\pi \cdot X \approx_\alpha^? \rho \cdot X$ is only true when X ranges over nominal terms, say s, for which all atoms in the difference set of π and ρ (i.e., the set $\{a : \pi(a) \neq \rho(a)\}$) are fresh in s.

If the support of a permutation π is fresh for X then $\pi \cdot X \approx_\alpha id \cdot X$. Thus a set of freshness constraints (i.e., a freshness context) can be used to specify that a permutation will have no effect on the instances of X. This is why in *nominal unification* [16], the solution for a problem is a pair consisting of a freshness context and a substitution.

The use of freshness contexts is natural when dealing with "syntactic" nominal unification, but in the presence of equational axioms (i.e., equational nominal unification) it is not straightforward. For example, in the case of C-nominal unification (nominal unification modulo commutativity), to specify that a permutation has no effect on the instances of X modulo C, in other words, to specify that the permutation does not affect a given C-equivalence class, we need something more than a freshness constraint (note that $(a\,b)(a+b) = b + a =_C a + b$, so the permutation $(a\,b)$ fixes the term $a + b$, despite the fact that a and b are not fresh).

In this paper, we propose to axiomatise α-equivalence of nominal terms using permutation fixed-point constraints: we write $\pi \curlywedge t$ (read "π fixes t") if t is a fixed-point of π. We show how to derive fixed-point constraints from primitive constraints of the form $\pi \curlywedge X$, and show the correctness of this approach by proving that the α-equivalence relation generated in this way coincides with the one axiomatised via freshness constraints. We then show how fixed-point constraints can be used to solve nominal unification problems modulo C.

In [4, 3, 2], the authors have proposed techniques to deal with α-equivalence modulo the equational theories A, C and AC using the standard approach via freshness constraints. The works [3, 2] show that despite the fact that C-unification problems have solutions generated by a finite family of fixed-point equations, there is no finitary representation of the admissible set of solutions using only freshness constraints and substitutions. Also, in [15] it is shown how nominal unification problems in a language with recursive let operators gives rise to solutions expressed in terms of freshness constraints and nominal fixed-point equations.

In this paper, we will develop an extension of fixed-point constraints modulo commutativity, namely, $\curlywedge_C$, and provide a set of rules for checking fixed-point judgements and α-equivalence judgements modulo C, which will provide a finitary representation of nominal C-unification solutions, consisting only of primitive fixed-point constraints and substitutions.

Overview

Section 2 presents the required preliminaries on nominal syntax. Section 3 introduces nominal α-equivalence using fixed-point constraints instead of freshness constraints. Section 4 introduces a sound and complete rule-based algorithm for nominal unification using fixed-point constraints. Before concluding, Section 5 shows how fixed-point constraints are used to finitely represent solutions of fixed-point equations, and so of nominal C-unification problems.

2   Preliminaries

We assume the reader is familiar with the notions of *nominal set* and *nominal syntax*. In this section we recall the main concepts and notations that are needed in this paper; for more details we refer the reader to [14, 16].

2.1   Nominal Terms

Let $\mathbb{A}$ be a fixed and countably infinite set of elements $a, b, c, \ldots$, which will be called *atoms* (atomic names). A permutation on $\mathbb{A}$ is a bijection on $\mathbb{A}$ with finite domain.

Fix a countably infinite set $\mathcal{X} = \{X, Y, Z, \ldots\}$ of variables and a countable set $\mathcal{F} = \{f, g, \ldots\}$ of function symbols.

▶ **Definition 1** (Nominal grammar). Nominal terms are generated by the following grammar.

$$s, t := a \mid [a]t \mid (t_1, \ldots, t_n) \mid f\ t \mid \pi \cdot X$$

where a is an *atom term*, $[a]t$ denotes the *abstraction* of the atom a over the term t, $(t_1, \ldots, t_n)$ is a tuple, $f\ t$ denotes the *application of* f *to* t and $\pi \cdot X$ is a *moderated variable* or *suspension*, where π is an atom permutation.

We follow the *permutative convention* [11, Convention 2.3] for atoms throughout the paper, i.e., atoms a, b, c range permutatively over $\mathbb{A}$ so that they are always pairwise different, unless stated otherwise.

Atom *permutations* are represented by finite lists of *swappings*, which are pairs of different atoms $(a\ b)$; hence, a permutation π is generated by the following grammar:

$$\pi := Id \mid (a\ b)\pi.$$

We call Id the identity permutation, which is usually omitted from the list of swappings defining a permutation. Suspensions of the form $Id \cdot X$ will be represented just by X. We write π^{-1} for the *inverse* of π, and use $\circ$ to denote the composition of permutations. For example, if $\pi = (a\ b)(b\ c)$ then $\pi(c) = a$ and $c = \pi^{-1}(a)$.

The *difference set* of two permutations π, π' is $\mathtt{ds}(\pi, \pi') = \{a \mid \pi(a) \neq \pi'(a)\}$.

We write $\mathtt{Var}(t)$ for the set of variables occurring in t. Ground terms are terms without variables, that is $\mathtt{Var}(t) = \emptyset$. A ground term may still contain atoms, for example a is a ground term and X is not.

▶ **Definition 2** (Permutation action). The action of a permutation π on a term t is defined by induction on the number of swappings in π:

$Id \cdot t = t$ and $((a\ b)\pi) \cdot t = (a\ b) \cdot (\pi \cdot t)$, where

$(a\ b) \cdot a = b,$
$(a\ b) \cdot b = a,$ $(a\ b) \cdot (\pi \cdot X) = ((a\ b) \circ \pi) \cdot X,$ $(a\ b) \cdot [c]t = [(a\ b) \cdot c](a\ b) \cdot t$
$(a\ b) \cdot c = c$ $(a\ b) \cdot f\ t = f\ (a\ b) \cdot t,$ $(a\ b) \cdot (t_1, \ldots, t_n) = ((a\ b) \cdot t_1, \ldots, (a\ b) \cdot t_n)$

▶ **Definition 3** (Substitution). *Substitutions* are generated by the grammar

$$\sigma ::= id \mid [X \mapsto s]\sigma.$$

Postfix notation is used for substitution application and $\circ$ for composition: $t(\sigma \circ \sigma') = (t\sigma)\sigma'$. Substitutions act on terms elementwise in the natural way: $t\ id = t$, $t[X \mapsto s]\sigma = (t[X \mapsto s])\sigma$, where

$$\begin{array}{ll} a[X \mapsto s] = a & (t_1, \ldots, t_n)[X \mapsto s] = (t_1[X \mapsto s], \ldots, t_n[X \mapsto s]) \\ (f\ t)[X \mapsto s] = f(t[X \mapsto s]) & (\pi \cdot X)[X \mapsto s] = \pi \cdot s \\ ([a]t)[X \mapsto s] = [a](t[X \mapsto s]) & (\pi \cdot Y)[X \mapsto s] = \pi \cdot Y \end{array}$$

2.2 Nominal sets and support

Let S be a set equipped with an action of the group $\texttt{Perm}(\mathbb{A})$ of finite permutations of $\mathbb{A}$.

▶ **Definition 4.** A set $A \subset \mathbb{A}$ is a *support* for an element $x \in S$ if for all $\pi \in \texttt{Perm}(\mathbb{A})$, the following holds

$$((\forall a \in A)\ \pi(a) = a) \Rightarrow \pi \cdot x = x \tag{1}$$

A *nominal set* is a set equipped with an action of the group $\texttt{Perm}(\mathbb{A})$, that is, a $\texttt{Perm}(\mathbb{A})$-set, all of whose elements have finite support.

As in [14], we denote by $\texttt{supp}_S(x)$ the least finite support of x, that is,

$$\texttt{supp}_S(x) := \bigcap \{A \in \mathcal{P}(\mathbb{A}) \mid A \text{ is a finite support for } x\}.$$

We write $\texttt{supp}(x)$ when S is clear from the context. Clearly, each $a \in \mathbb{A}$ is finitely supported by $\{a\}$, therefore $\texttt{supp}(a) = \{a\}$.

3 Constraints

The native notion of equality on nominal terms is α-equivalence, written $s \approx_\alpha t$. This relation is usually axiomatised using a *freshness relation* between atoms and terms, written $a \# t$ – read "a fresh for t", which, intuitively, corresponds to the idea of an atom not occurring free in a term (see for instance [16, 8]). However, freshness is not a primitive notion in nominal sets; it is derived using the quantifier $\mathsf{И}$ combined with a notion of fixed-point, as shown by Pitts [14]:

$$a \# X \Leftrightarrow \mathsf{И} a'.(a\ a') \cdot X = X.$$

In this work, instead of defining α-equivalence using freshness, we define it using the more primitive notion of *fixed-point* under the action of permutations. We will denote this relation $\overset{\curlywedge}{\approx}_\alpha$, and show that it coincides with $\approx_\alpha$ on ground terms, i.e., the relation defined using fixed-points of permutations corresponds to the relation defined using freshness. For non-ground terms, there is also a correspondence, but under different kinds of assumptions (fixed-point constraints vs. freshness constraints).

3.1 Fixed-points of permutations and term equality

We start by defining a binary relation that describes which elements of a nominal set S are fixed-points of a permutation $\pi \in \texttt{Perm}(\mathbb{A})$:

▶ **Definition 5** (Fixed-point relation). Let S be a nominal set. The *fixed-point relation* $\curlywedge \subseteq \texttt{Perm}(\mathbb{A}) \times S$ is defined as: $\pi \curlywedge x \Leftrightarrow dom(\pi) \cap \texttt{supp}(x) = \emptyset$. Read "$\pi \curlywedge x$" as "$\pi$ *fixes* x".

The fixed-point relation between permutations and terms will play an important role in the definition of α-equality. Below we define the *fixed-point* constraints and *equality* constraints using predicates $\curlywedge$ and $\overset{\curlywedge}{\approx}_\alpha$ and then give deduction rules to derive fixed-point and equality judgements. Intuitively,

-   $s \overset{\curlywedge}{\approx}_\alpha t$ will mean that s and t are α-equivalent, i.e., equivalent modulo renaming of abstracted atoms.

$$\frac{\pi(a) = a}{\Upsilon \vdash \pi \curlywedge a} \,(\curlywedge\mathbf{a}) \qquad\qquad \frac{\mathrm{supp}(\pi^{\pi'^{-1}}) \subseteq \mathrm{supp}(\mathrm{perm}(\Upsilon|_X))}{\Upsilon \vdash \pi \curlywedge \pi' \cdot X} \,(\curlywedge\mathbf{var})$$

$$\frac{\Upsilon \vdash \pi \curlywedge t}{\Upsilon \vdash \pi \curlywedge \mathsf{f}\, t} \,(\curlywedge\mathsf{f}) \qquad\qquad \frac{\Upsilon \vdash \pi \curlywedge t_1 \quad \ldots \quad \Upsilon \vdash \pi \curlywedge t_n}{\Upsilon \vdash \pi \curlywedge (t_1, \ldots, t_n)} \,(\curlywedge\mathbf{tuple})$$

$$\frac{\Upsilon, (c_1\ c_2) \curlywedge \mathtt{Var}(t) \vdash \pi \curlywedge (a\ c_1) \cdot t}{\Upsilon \vdash \pi \curlywedge [a]t} \,(\curlywedge\mathbf{abs}), \quad \begin{array}{l} c_1 \text{ and } c_2 \\ \text{new names} \end{array}$$

Figure 1 Fixed-point rules.

- $\pi \curlywedge t$ will mean that the permutation π fixes the nominal term t, that is, $\pi \cdot t \stackrel{\curlywedge}{\approx}_\alpha t$. This means that π has "no effect" on t except for the renaming of bound names, for instance, $(a\ b) \curlywedge [a]a$ but not $(a\ b) \curlywedge \mathsf{f}\, a$.

▶ **Definition 6** (Fixed-point and equality constraints). A *fixed-point constraint* is a pair $\pi \curlywedge t$ of a permutation π and a term t. An *α-equivalence constraint* is a pair of the form $s \stackrel{\curlywedge}{\approx}_\alpha t$. We call a fixed-point constraint of the form $\pi \curlywedge X$ a *primitive fixed-point constraint* and a set of such constraints is called a *fixed-point context*. $\Upsilon, \Psi, \ldots$ range over fixed-point contexts. We write $\pi \curlywedge \mathtt{Var}(t)$ as an abbreviation for the set of constraints $\{\pi \curlywedge X \mid X \in \mathtt{Var}(t)\}$.

The set of variables $\mathtt{Var}(\Upsilon)$ is defined as expected. The set of permutations of a fixed-point context Υ with respect to the variable $X \in \mathtt{Var}(\Upsilon)$, denoted by $\mathtt{perm}(\Upsilon|_X)$, is defined as $\mathtt{perm}(\Upsilon|_X) := \{\pi \mid \pi \curlywedge X \in \Upsilon\}$. For a substitution σ and a fixed-point context Υ we define $\Upsilon\sigma := \{\pi \curlywedge X\sigma \mid \pi \curlywedge X \in \Upsilon\}$.

To define the relation $\curlywedge$, we rely on the notion of *conjugation* of permutations. The conjugate of π with respect to ρ, denoted as π^ρ, is the result of the composition: $\rho \circ \pi \circ \rho^{-1}$.

$$\begin{array}{ccccccccc} \pi^\rho: & A & \stackrel{\rho^{-1}}{\to} & A & \stackrel{\pi}{\to} & A & \stackrel{\rho}{\to} & A \\ & a & \mapsto & \rho^{-1}(a) & \mapsto & \pi(\rho^{-1}(a)) & \mapsto & \rho(\pi(\rho^{-1}(a))) \end{array}$$

▶ **Definition 7** (Judgements). A *fixed-point judgement* is a tuple $\Upsilon \vdash \pi \curlywedge t$ of a fixed-point context and a fixed-point constraint. An *α-equivalence judgement* is a tuple $\Psi \vdash s \stackrel{\curlywedge}{\approx}_\alpha t$ of a fixed-point context and an equality constraint. The derivable fixed-point and α-equivalence judgements are defined by the rules in Figures 1 and 2.

▶ **Example 8.** The term $[a]\mathsf{f}a$ is a fixed-point for the permutation $(a\ b)$, since $(a\ b)[a]\mathsf{f}a \approx_\alpha [b]\mathsf{f}b$, therefore, $(a\ b) \curlywedge [a]\mathsf{f}a$. However, $\mathsf{f}a$ is not a fixed-point for $(a\ b)$, since $(a\ b) \cdot \mathsf{f}a \not\approx_\alpha \mathsf{f}b$.

Rule $(\curlywedge\mathbf{a})$ states that if $a \notin dom(\pi)$, then a is a fixed-point of π.

In rule $(\curlywedge\mathbf{var})$, the condition $\mathrm{supp}(\pi^{\pi'^{-1}}) \subseteq \mathrm{supp}(\mathrm{perm}(\Upsilon|_X))$ means that the permutation can be generated from $\mathrm{perm}(\Upsilon|_X)$, hence it fixes X. Rules $(\curlywedge\mathsf{f})$ and $(\curlywedge\mathbf{tuple})$ are straightforward. Rule $(\curlywedge\mathbf{abs})$ is the most interesting one. The intuition behind this rule is the following: $[a]t$ is a fixed-point of π if $\pi \cdot [a]t$ is α-equivalent to $[a]t$, that is, $[\pi(a)]\pi \cdot t$ is α-equivalent to $[a]t$; the latter means that the only atom that could be affected by π is a, hence, if we replace occurrences of a in t with another, new atom c_1, π should have no effect.

The α-equality relation is defined in terms of fixed-point constraints. Rules $(\stackrel{\curlywedge}{\approx}_\alpha \mathsf{a})$, $(\stackrel{\curlywedge}{\approx}_\alpha \mathsf{f})$, $(\stackrel{\curlywedge}{\approx}_\alpha [\mathsf{a}])$ and $(\stackrel{\curlywedge}{\approx}_\alpha \mathsf{tuple})$ are defined as expected, whereas the intuition behind rule $(\stackrel{\curlywedge}{\approx}_\alpha \mathsf{var})$ is similar to the corresponding rule in Figure 1. The most interesting rule is $(\stackrel{\curlywedge}{\approx}_\alpha \mathsf{ab})$.

$$\frac{}{\Upsilon \vdash a \stackrel{\curlywedge}{\approx}_\alpha a} \; (\stackrel{\curlywedge}{\approx}_\alpha \mathbf{a}) \qquad \frac{\mathtt{supp}((\pi')^{-1} \circ \pi) \subseteq \mathtt{supp}(\mathtt{perm}(\Upsilon|_X))}{\Upsilon \vdash \pi \cdot X \stackrel{\curlywedge}{\approx}_\alpha \pi' \cdot X} \; (\stackrel{\curlywedge}{\approx}_\alpha \mathbf{var})$$

$$\frac{\Upsilon \vdash t \stackrel{\curlywedge}{\approx}_\alpha t'}{\Upsilon \vdash \mathtt{f}\, t \stackrel{\curlywedge}{\approx}_\alpha \mathtt{f}\, t'} \; (\stackrel{\curlywedge}{\approx}_\alpha \mathtt{f}) \qquad \frac{\Upsilon \vdash t_1 \stackrel{\curlywedge}{\approx}_\alpha t_1' \quad \ldots \quad \Upsilon \vdash t_n \stackrel{\curlywedge}{\approx}_\alpha t_n'}{\Upsilon \vdash (t_1, \ldots, t_n) \stackrel{\curlywedge}{\approx}_\alpha (t_1', \ldots, t_n')} \; (\stackrel{\curlywedge}{\approx}_\alpha \mathbf{tuple})$$

$$\frac{\Upsilon \vdash t \stackrel{\curlywedge}{\approx}_\alpha t'}{\Upsilon \vdash [a]t \stackrel{\curlywedge}{\approx}_\alpha [a]t'} \; (\stackrel{\curlywedge}{\approx}_\alpha [\mathbf{a}]) \qquad \frac{\Upsilon \vdash s \stackrel{\curlywedge}{\approx}_\alpha (a\ b) \cdot t \quad \Upsilon, (c_1\ c_2) \curlywedge \mathtt{Var}(t) \vdash (a\ c_1) \curlywedge t}{\Upsilon \vdash [a]s \stackrel{\curlywedge}{\approx}_\alpha [b]t} \; (\stackrel{\curlywedge}{\approx}_\alpha \mathbf{ab})$$

🟨 **Figure 2** Rules for equality. In rule $(\stackrel{\curlywedge}{\approx}_\alpha \mathbf{ab})$, c_1 and c_2 are new names.

Intuitively, it states that for two abstractions $[a]s$ and $[b]t$ to be equivalent, we must obtain equivalent terms if we rename in one of them, in our case t, the abstracted atom b to a, so that they both use the same atom. Moreover, the atom a should not occur free in t, which is checked by stating that $(a\ c_1)$ fixes t for some new atom c_1 that is not in the support of the variables occurring in t.

We prove below that $\stackrel{\curlywedge}{\approx}_\alpha$ is indeed an equivalence relation, for which we need to study the properties of the relations $\stackrel{\curlywedge}{\approx}_\alpha$ and $\curlywedge$, starting with *inversion* and *equivariance*.

▶ **Lemma 9** (Inversion). *The inference rules for* $\stackrel{\curlywedge}{\approx}_\alpha$ *are invertible.*

The notion of *equivariance* relies on the conjugation of the permutation π by ρ, π^ρ. The following basic property is used in the proofs in this section.

▶ **Lemma 10.** *Let* ρ *be a permutation in* $\mathtt{Perm}(\mathbb{A})$ *and* a, b *atoms in* $\mathbb{A}$. *Then* $(a\ b)^\rho = (\rho(a)\ \rho(b))$.

▶ **Lemma 11.**
  i.) $\Upsilon \vdash \pi \curlywedge t$ *if and only if* $\mathtt{supp}(\pi) \cap \mathtt{supp}(t) = \emptyset$.
  ii.) *If* $\Upsilon \vdash s \stackrel{\curlywedge}{\approx}_\alpha t$ *then* $\mathtt{supp}(s) = \mathtt{supp}(t)$.

Proof. Both parts are proved by induction. In part (i), we analyse cases depending on the last rule applied in the derivation of $\Upsilon \vdash \pi \curlywedge t$. We show the cases for rules $(\curlywedge\mathbf{var})$ and $(\curlywedge\mathbf{abs})$, the other cases follow directly by induction.

If the last rule applied is $(\curlywedge\mathbf{var})$ then $t = \pi' \cdot X$ and $\Upsilon \vdash \pi \curlywedge \pi' \cdot X$ if and only if (Inversion Lemma) $\mathtt{supp}(\pi^{\pi'^{-1}}) \subseteq \mathtt{supp}(\mathtt{perm}(\Upsilon|_X))$, if and only if $\mathtt{supp}(\pi) \subseteq \pi' \cdot \mathtt{supp}(\mathtt{perm}(\Upsilon|_X))$. Since $\mathtt{supp}(X) \cap \mathtt{supp}(\mathtt{perm}(\Upsilon|_X)) = \emptyset$ by Definition 5, we deduce $\mathtt{supp}(\pi) \cap \mathtt{supp}(\pi' \cdot X) = \emptyset$ as required.

If the last rule applied is $(\curlywedge\mathbf{abs})$, then $t = [a]t'$ and $\Upsilon \vdash \pi \curlywedge [a]t'$ if and only if (Inversion Lemma) $\Upsilon, (c_1\ c_2) \curlywedge \mathtt{Var}(t') \vdash \pi \curlywedge (a\ c_1) \cdot t'$. By induction, $\mathtt{supp}(\pi) \cap \mathtt{supp}((a\ c_1)t') = \emptyset$ and since $\mathtt{supp}([a]t') = \mathtt{supp}((a\ c_1) \cdot t') - \{c_1\}$ (because c_1 is a new atom and $(c_1\ c_2) \curlywedge \mathtt{Var}(t')$), we obtain $\mathtt{supp}(\pi) \cap \mathtt{supp}([a]t') = \emptyset$ as required.

The proof for part (ii), by induction on the derivation of $\Upsilon \vdash s \stackrel{\curlywedge}{\approx}_\alpha t$, is similar. In the case of rule $(\stackrel{\curlywedge}{\approx}_\alpha \mathbf{var})$, the premise implies that $\mathtt{ds}(\pi, \pi') \cap \mathtt{supp}(X) = \emptyset$, hence $\mathtt{supp}(\pi \cdot X) = \mathtt{supp}(\pi' \cdot X)$. In the case of rule $(\stackrel{\curlywedge}{\approx}_\alpha \mathbf{ab})$, by induction hypothesis $\mathtt{supp}(s) = \mathtt{supp}((a\ b) \cdot t)$ and since we know that $(a\ c_1) \curlywedge t$, using part 1 we obtain the result. ◀

▶ **Lemma 12** (Equivariance).
  i.) $\Upsilon \vdash \pi \curlywedge t$ iff $\Upsilon \vdash \pi^\rho \curlywedge \rho \cdot t$, for any permutation ρ.
 ii.) If $\Upsilon \vdash s \overset{\curlywedge}{\approx}_\alpha t$ then $\Upsilon \vdash \pi \cdot s \overset{\curlywedge}{\approx}_\alpha \pi \cdot t$.

Proof. By induction on the rules of Figures 1 and 2.   ◀

▶ **Lemma 13** ($\curlywedge$ preservation under $\overset{\curlywedge}{\approx}_\alpha$). If $\Upsilon \vdash s \overset{\curlywedge}{\approx}_\alpha t$ and $\Upsilon \vdash \pi \curlywedge s$ then $\Upsilon \vdash \pi \curlywedge t$.

Proof. Direct consequence of Lemma 11.   ◀

▶ **Proposition 14** (Strengthening for $\curlywedge$). If $\Upsilon, \pi \curlywedge X \vdash \pi' \curlywedge s$ and $\mathtt{supp}(\pi) \subseteq \mathtt{supp}(\mathtt{perm}(\Upsilon|_X))$ or $X \notin \mathtt{Var}(s)$ then $\Upsilon \vdash \pi' \curlywedge s$.

▶ **Proposition 15** (Strengthening for $\overset{\curlywedge}{\approx}_\alpha$). If $\Upsilon, \pi \curlywedge X \vdash s \overset{\curlywedge}{\approx}_\alpha t$ and
$\mathtt{supp}(\pi) \subseteq \mathtt{supp}(\mathtt{perm}(\Upsilon|_X))$ or $X \notin \mathtt{Var}(s,t)$, then $\Upsilon \vdash s \overset{\curlywedge}{\approx}_\alpha t$.

▶ **Proposition 16** (Weakening). Suppose that $\Upsilon \vdash \Upsilon'\sigma$. Then,
1. $\Upsilon' \vdash \pi \curlywedge s \Longrightarrow \Upsilon \vdash \pi \curlywedge s\sigma$.
2. $\Upsilon' \vdash s \overset{\curlywedge}{\approx}_\alpha t \Longrightarrow \Upsilon \vdash s\sigma \overset{\curlywedge}{\approx}_\alpha t\sigma$.

Proof. By induction on the rules of Figures 1 and 2.   ◀

▶ **Example 17.** Notice that $(a\ c) \curlywedge X \vdash (a\ b) \curlywedge (b\ c) \cdot X$, for

$$(a\ c) \curlywedge X \vdash (a\ b)^{(b\ c)} \curlywedge X \Leftrightarrow (a\ c) \curlywedge X \vdash (a\ c) \curlywedge X \quad \text{(by Equivariance)} \tag{2}$$

The following correctness property states that $\curlywedge$ is indeed the fixed-point relation:

▶ **Theorem 18.** Let Υ, π and t be a fixed-point context, a permutation and a nominal term, respectively. $\Upsilon \vdash \pi \curlywedge t$ iff $\Upsilon \vdash \pi \cdot t \overset{\curlywedge}{\approx}_\alpha t$.

Sketch. In both directions the proof follows by induction on the structure of the term t and by case analysis on the last rule applied in the derivation. We show only $\Upsilon \vdash \pi \curlywedge t \Rightarrow \Upsilon \vdash \pi \cdot t \overset{\curlywedge}{\approx}_\alpha t$. Below we sketch the interesting cases, the other cases follow by induction hypothesis easily.
1. The last rule is ($\curlywedge$**var**). In this case, $t = \pi' \cdot X$ and $\mathtt{supp}((\pi')^{-1} \circ \pi \circ \pi') \subseteq \mathtt{supp}(\mathtt{perm}(\Upsilon|_X))$
  and therefore, $\pi \cdot (\pi' \cdot X) \overset{\curlywedge}{\approx}_\alpha \pi' \cdot X$, via rule ($\overset{\curlywedge}{\approx}_\alpha$ **var**).
2. The last rule is ($\curlywedge$**abs**). In this case, $t = [a]t'$ and $\pi \curlywedge t$ has a derivation of the form:
$$\frac{\begin{array}{c}\Pi \\ \hline \Upsilon, (c_1\ c_2) \curlywedge \mathtt{Var}(t') \vdash \pi \curlywedge (a\ c_1) \cdot t'\end{array}}{\Upsilon \vdash \pi \curlywedge [a]t'}$$
From $\Upsilon, (c_1\ c_2) \curlywedge \mathtt{Var}(t') \vdash \pi \curlywedge (a\ c_1) \cdot t'$ it follows, from Lemma 11:

$$\mathtt{supp}(\pi) \cap \mathtt{supp}((a\ c_1) \cdot t') = \emptyset. \tag{3}$$

We need to prove that $\Upsilon \vdash [\pi(a)]\pi \cdot t' \overset{\curlywedge}{\approx}_\alpha [a]t'$, that is, $\Upsilon \vdash \pi \cdot t' \overset{\curlywedge}{\approx}_\alpha (\pi(a)\ a) \cdot t'$ and also $\Upsilon, (c_1\ c_2) \curlywedge \mathtt{Var}(t') \vdash (\pi(a)\ c_1) \curlywedge t'$ for some new atoms c_1, c_2.
By IH, there exist a proof Π' for $\Upsilon, (c_1\ c_2) \curlywedge \mathtt{Var}(t') \vdash \pi \cdot ((a\ c_1).t') \overset{\curlywedge}{\approx}_\alpha (a\ c_1) \cdot t'$. Let $\Upsilon' = \Upsilon, (c_1\ c_2) \curlywedge \mathtt{Var}(t')$. The following equivalence holds:

$$\Upsilon' \vdash \pi.((a\ c_1) \cdot t') \overset{\curlywedge}{\approx}_\alpha (a\ c_1) \cdot t' \iff \Upsilon' \vdash (\pi(a)\ c_1) \cdot (\pi \cdot t') \overset{\curlywedge}{\approx}_\alpha (a\ c_1) \cdot t' \tag{4}$$

Also, $\Upsilon' \vdash (\pi \cdot t') \overset{\curlywedge}{\approx}_\alpha (\pi(a)\ c_1) \cdot ((a\ c_1) \cdot t')$ by Equivariance. And since $\Upsilon' \vdash (\pi(a)\ c_1) \cdot ((a\ c_1) \cdot t') \overset{\curlywedge}{\approx}_\alpha (\pi(a)\ a) \cdot t'$, we are done.   ◀

$$\frac{}{\Delta \vdash a\#b}\,(\#\mathbf{a}) \qquad\qquad\qquad \frac{\pi^{-1}(a)\#X \in \Delta}{\Delta \vdash a\#\pi' \cdot X}\,(\#\mathbf{var})$$

$$\frac{\Delta \vdash a\#t}{\Delta \vdash a\#\mathsf{f}\ t}\,(\#\mathbf{f}) \qquad\qquad \frac{\Delta \vdash a\#t_1 \quad \ldots \quad \Delta \vdash a\#t_n}{\Delta \vdash a\#(t_1,\ldots,t_n)}\,(\#\mathbf{tuple})$$

$$\frac{}{\Delta \vdash a\#[a]t}\,(\#[\mathbf{a}]) \qquad\qquad\qquad \frac{\Delta \vdash a\#t}{\Upsilon \vdash a\#[b]t}\,(\#\mathbf{abs})$$

▨ **Figure 3** Rules for freshness.

$$\frac{}{\Delta \vdash a \approx_\alpha a}\,(\approx_\alpha \mathbf{a}) \qquad\qquad \frac{\mathtt{ds}(\pi,\pi')\#X \subseteq \Delta}{\Delta \vdash \pi \cdot X \approx_\alpha \pi' \cdot X}\,(\approx_\alpha \mathbf{var})$$

$$\frac{\Delta \vdash t \approx_\alpha t'}{\Delta \vdash \mathsf{f}\ t \approx_\alpha \mathsf{f}\ t'}\,(\approx_\alpha \mathsf{f}) \qquad \frac{\Delta \vdash t_1 \approx_\alpha t_1' \quad \ldots \quad \Delta \vdash t_n \approx_\alpha t_n'}{\Delta \vdash (t_1,\ldots,t_n) \approx_\alpha (t_1',\ldots,t_n')}\,(\approx_\alpha \mathbf{tuple})$$

$$\frac{\Delta \vdash t \approx_\alpha t'}{\Delta \vdash [a]t \approx_\alpha [a]t'}\,(\approx_\alpha [\mathbf{a}]) \qquad \frac{\Delta \vdash\ s \approx_\alpha (a\ b).t \quad \Delta \vdash a\#t}{\Upsilon \vdash [a]s \approx_\alpha [b]t}\,(\approx_\alpha \mathbf{ab})$$

▨ **Figure 4** Rules for α-equality via freshness.

3.2 From freshness to fixed-point constraints

In this section we show that the α-equivalence relation defined in terms of *freshness constraints*, denoted as $\approx_\alpha$, is equivalent to $\overset{\curlywedge}{\approx}_\alpha$, given that a transformation $[\_]^\curlywedge$ from freshness to fixed-point constraints and a transformation $[\_]^\#$ from fixed-point to freshness constraints can be defined. In the standard approach [13, 8], the freshness relation $(a\#t)$ and the α-equivalence relation $s \approx_\alpha t$ (w.r.t. $\#$), are axiomatised using the rules in Figures 3 and 4, respectively.

To define $\approx_\alpha$ we use the *difference set* of two permutations in rule $(\approx_\alpha \mathbf{var})$, and $\mathtt{ds}(\pi,\pi')\#X = \{a\#X \mid a \in \mathtt{ds}(\pi,\pi')\}$.

The symbols Δ and ∇ denote *freshness contexts*, that is, sets of freshness constraints of the form $a\#X$, meaning that a is fresh in X. The domain of a freshness context Δ, denoted by $\mathtt{dom}(\Delta)$, consists of the atoms occurring in Δ; $\Delta|_X$ consists of the restriction of Δ to the freshness constraints on variable X, that is, the set $\{a\#X \mid a\#X \in \Delta\}$. Below we denote by $\mathfrak{F}_\#$ the family of freshness contexts, and by $\mathfrak{F}_\curlywedge$ the family of fixed-point contexts. The mapping $[\_]_\curlywedge$ below associates each freshness constraint in Δ with a fixed-point constraint:

$$[\_]_\curlywedge :\quad \begin{array}{ccl} \Delta & \longrightarrow & \mathfrak{F}_\curlywedge \\ a\#X & \mapsto & (a\ c_a) \curlywedge X \text{ where } c_a \text{ is a new name.} \end{array}$$

We denote by $[\Delta]_\curlywedge$ the image of Δ under $[\_]_\curlywedge$.

The mapping $[\_]_\curlywedge$ below associates each fixed-point constraint in Υ with a freshness constraint:

$$[\_]_\# :\quad \begin{array}{ccl} \Upsilon & \longrightarrow & \mathfrak{F}_\# \\ \pi \curlywedge X & \mapsto & \mathtt{supp}(\pi)\#X. \end{array}$$

We denote by $[\Upsilon]_\#$ the image of Υ under $[\_]_\#$.

▶ **Lemma 19.**
1. $\Delta \vdash a\#t \Leftrightarrow [\Delta]_{\curlywedge}, (c_2\ c_1) \curlywedge \mathtt{Var}(t) \vdash (a\ c_1) \curlywedge t.$
2. $\Upsilon \vdash \pi \curlywedge t \Leftrightarrow [\Upsilon]_{\#} \vdash \overline{\mathtt{supp}(\pi)\#t}.$

▶ **Theorem 20.** $\overset{\curlywedge}{\approx}_\alpha$ *coincides with* $\approx_\alpha$ *on ground terms, that is,* $\vdash s \approx_\alpha t \iff \vdash s \overset{\curlywedge}{\approx}_\alpha t.$
More generally,
1. $\Delta \vdash s \approx_\alpha t \Rightarrow [\Delta]_{\curlywedge} \vdash s \overset{\curlywedge}{\approx}_\alpha t.$
2. $\Upsilon \vdash s \overset{\curlywedge}{\approx}_\alpha t \Rightarrow [\Upsilon]_{\#} \vdash s \approx_\alpha t.$

Sketch.
1. The proof is by induction on the derivation of $\Delta \vdash s \approx_\alpha t$. The interesting case is when the derivation is an instance of ($\approx_\alpha$ **var**):
$$\frac{\mathtt{ds}(\pi, \pi_1)\#X \subseteq \Delta}{\Delta \vdash \pi \cdot X \approx_\alpha \pi_1 \cdot X}\ (\approx_\alpha\ \mathbf{var})$$
We want to show that $[\Delta]_{\curlywedge} \vdash \pi \cdot X \overset{\curlywedge}{\approx}_\alpha \pi_1 \cdot X$. To use rule ($\overset{\curlywedge}{\approx}_\alpha$ **var**), we need to show that $\mathtt{supp}(\pi_1^{-1} \circ \pi) \subseteq \mathtt{supp}(\mathtt{perm}(([\Delta]\curlywedge)|_X))$. Let $b \in \mathtt{supp}(\pi_1^{-1} \circ \pi)$ and suppose $b \notin \mathtt{ds}(\pi, \pi_1)$. Then $\pi(b) = \pi_1(b)$ and $\pi_1^{-1}(\pi(b)) = b$, contradiction. Therefore, $b \in \mathtt{ds}(\pi, \pi_1)$ and $(b\ c_b) \curlywedge X \in [\Delta]_{\curlywedge}$ (for c_b a new name), and the result follows. ◀

As a corollary, since $\approx_\alpha$ is an equivalence relation [16], we deduce that $\overset{\curlywedge}{\approx}_\alpha$ is also an equivalence relation.

▶ **Theorem 21.** $\overset{\curlywedge}{\approx}_\alpha$ *is an equivalence relation.*

4 Nominal Unification via fixed-point constraints

In this section we define the notion of nominal unification in terms of fixed-point constraints.

▶ **Definition 22.** A *unification problem* $\mathtt{Pr}$ consists of a finite set of equations and fixed-point constraints of the form $s \overset{\curlywedge}{\approx}_\alpha^{?} t$ and $\pi \curlywedge^{?} t$, respectively.

We design a unification algorithm via the simplification rules presented in Table 1. These rules act on unification problems $\mathtt{Pr}$. We abbreviate $(t_1, \ldots, t_n)$ as $(\widetilde{t})_n$, and for a set S, $\overline{\pi \curlywedge S} = \{\pi \curlywedge X \mid X \in S\}$.

We write $\mathtt{Pr} \Longrightarrow \mathtt{Pr'}$, when $\mathtt{Pr'}$ is obtained from $\mathtt{Pr}$ by applying a simplification rule from Table 1 and we write $\overset{*}{\Longrightarrow}$ for the reflexive and transitive closure of $\Longrightarrow$.

▶ **Lemma 23.** *There is no infinite chain of reductions* $\Longrightarrow$ *starting from a problem* $\mathtt{Pr}$.

Proof. Termination of the simplification rules follows directly from the fact that the following measure of the size of $\mathtt{Pr}$ is strictly decreasing:
$[\mathtt{Pr}] = (n_1, M)$ where n_1 is the number of different variables used in $\mathtt{Pr}$, and M is the multiset of sizes of equality constraints and non-primitive fixed-point constraints occurring in $\mathtt{Pr}$.

Each simplification step either eliminates one variable (when an instantiation rule is used) and therefore decreases the first component of the interpretation, or leaves the first component unchanged but replaces a constraint with smaller ones and/or primitive ones. ◀

The normal form of $\mathtt{Pr}$ by $\Longrightarrow$ is defined as expected and denoted by $\langle \mathtt{Pr} \rangle_{\mathtt{nf}}$.

We say that an equality constraint $s \overset{\curlywedge}{\approx}_\alpha^{?} t$ is *reduced* when one of the following holds:
1. $s := a$ and $t := b$ are distinct atoms;
2. s and t are headed with different function symbols, that is, $s := \mathsf{f}\ s'$ and $t := \mathsf{g}\ t'$;

■ **Table 1** Simplification Rules for Problems. In ($\curlywedge abs$) and ($\overset{\curlywedge}{\approx}_\alpha abs2$), c_1 and c_2 are new names.

($\curlywedge at$)	$\text{Pr} \uplus \{\pi \curlywedge^? a\}$	$\Longrightarrow$ Pr, if $\pi(a) = a$
($\curlywedge f$)	$\text{Pr} \uplus \{\pi \curlywedge^? \mathsf{f}t\}$	$\Longrightarrow \text{Pr} \cup \{\pi \curlywedge^? t\}$
($\curlywedge t$)	$\text{Pr} \uplus \{\pi \curlywedge^? (\widetilde{t})_n\}$	$\Longrightarrow \text{Pr} \cup \{\pi \curlywedge^? t_1, \ldots, \pi \curlywedge^? t_n\}$
($\curlywedge abs$)	$\text{Pr} \uplus \{\pi \curlywedge^? [a]t\}$	$\Longrightarrow \text{Pr} \cup \{\pi \curlywedge^? (a\ c_1) \cdot t, \overline{(c_1\ c_2) \curlywedge^? \text{Var}(t)}\}$
($\curlywedge var$)	$\text{Pr} \uplus \{\pi \curlywedge^? \pi' \cdot X\}$	$\Longrightarrow \text{Pr} \cup \{\pi^{(\pi')^{-1}} \curlywedge^? X\}$, if $\pi' \neq Id$
($\overset{\curlywedge}{\approx}_\alpha a$)	$\text{Pr} \uplus \{a \overset{\curlywedge^?}{\approx}_\alpha a\}$	$\Longrightarrow$ Pr
($\overset{\curlywedge}{\approx}_\alpha f$)	$\text{Pr} \uplus \{\mathsf{f}\ t \overset{\curlywedge^?}{\approx}_\alpha \mathsf{f}\ t'\}$	$\Longrightarrow \text{Pr} \cup \{t \approx_\alpha^? t'\}$
($\overset{\curlywedge}{\approx}_\alpha t$)	$\text{Pr} \uplus \{(\widetilde{t})_n \approx_\alpha^? (\widetilde{t'})_n\}$	$\Longrightarrow \text{Pr} \cup \{t_1 \overset{\curlywedge^?}{\approx}_\alpha t_1', \ldots, t_n \overset{\curlywedge^?}{\approx}_\alpha t_n'\}$
($\overset{\curlywedge}{\approx}_\alpha abs1$)	$\text{Pr} \uplus \{[a]t \overset{\curlywedge^?}{\approx}_\alpha [a]t'\}$	$\Longrightarrow \text{Pr} \cup \{t \overset{\curlywedge^?}{\approx}_\alpha t'\}$
($\overset{\curlywedge}{\approx}_\alpha abs2$)	$\text{Pr} \uplus \{[a]t \overset{\curlywedge^?}{\approx}_\alpha [b]s\}$	$\Longrightarrow \text{Pr} \cup \{t \overset{\curlywedge^?}{\approx}_\alpha (a\ b) \cdot s, (a\ c_1) \curlywedge^? s, \overline{(c_1\ c_2) \curlywedge^? \text{Var}(s)}\}$
($\approx_\alpha var$)	$\text{Pr} \uplus \{\pi \cdot X \overset{\curlywedge^?}{\approx}_\alpha \pi' \cdot X\} \Longrightarrow \text{Pr} \cup \{(\pi')^{-1} \circ \pi \curlywedge^? X\}$	
($\approx_\alpha inst1$)	$\text{Pr} \uplus \{\pi \cdot X \overset{\curlywedge^?}{\approx}_\alpha t\}$	$\overset{[X \mapsto \pi^{-1}.t]}{\Longrightarrow} \text{Pr}\{X \mapsto \pi^{-1}.t\}$, if $X \notin \text{Var}(t)$
($\approx_\alpha inst2$)	$\text{Pr} \uplus \{t \overset{\curlywedge^?}{\approx}_\alpha \pi \cdot X\}$	$\overset{[X \mapsto \pi^{-1}.t]}{\Longrightarrow} \text{Pr}\{X \mapsto \pi^{-1}.t\}$, if $X \notin \text{Var}(t)$

3. s and t have different term constructors, that is, $s = [a]s'$ and $t = \mathsf{f}\ t'$, for some term former f, or $s = \pi \cdot X$ and $t = a$, etc.

A fixed-point constraint $\pi \curlywedge^? s$ is *reduced* when it is of the form $\pi \curlywedge^? a$ and $\pi(a) \neq a$, or $\pi \curlywedge^? X$, the former is called *inconsistent* whereas the latter is called *consistent*.

▶ **Example 24.** For $\text{Pr} = [a]\mathsf{f}(X, a) \overset{\curlywedge^?}{\approx}_\alpha [b]\mathsf{f}((b\ c) \cdot W, (a\ c) \cdot Y)$, we obtain the following derivation chain:

$$[a]\mathsf{f}(X, a) \overset{\curlywedge^?}{\approx}_\alpha [b]\mathsf{f}((b\ c) \cdot W, (a\ c) \cdot Y) \Longrightarrow \left\{ \begin{array}{l} \mathsf{f}(X, a) \overset{\curlywedge^?}{\approx}_\alpha \mathsf{f}((a\ b) \circ (b\ c) \cdot W, (a\ b) \circ (a\ c) \cdot Y), \\ (a\ c_1) \curlywedge^? \mathsf{f}((b\ c) \cdot W, (a\ c) \cdot Y), \\ (c_2\ c_1) \curlywedge^? W, (c_2\ c_1) \curlywedge^? Y \end{array} \right\}$$

$$\Longrightarrow \left\{ \begin{array}{l} X \overset{\curlywedge^?}{\approx}_\alpha (a\ b) \circ (b\ c) \cdot W, a \overset{\curlywedge^?}{\approx}_\alpha (a\ b) \circ (a\ c) \cdot Y, \\ (a\ c_1) \curlywedge^? (b\ c) \cdot W, (a\ c_1) \curlywedge^? (a\ c) \cdot Y, (c_2\ c_1) \curlywedge^? W, (c_2\ c_1) \curlywedge^? Y \end{array} \right\}$$

$$\overset{[Y \mapsto b]}{\Longrightarrow} \left\{ X \overset{\curlywedge^?}{\approx}_\alpha (a\ b) \circ (b\ c) \cdot W, (a\ c_1)^{(b\ c)} \curlywedge^? W, (a\ c_1) \curlywedge^? b, (c_2\ c_1) \curlywedge^? W, (c_2\ c_1) \curlywedge^? b \right\}$$

$$\overset{*}{\Longrightarrow} \left\{ X \overset{\curlywedge^?}{\approx}_\alpha (a\ b) \circ (b\ c) \cdot W, (a\ c_1) \curlywedge^? W, (c_2\ c_1) \curlywedge^? W \right\}$$

$$\overset{[X \mapsto (a\ b) \circ (b\ c) \cdot W]}{\Longrightarrow} \left\{ (a\ c_1) \curlywedge^? W, (c_2\ c_1) \curlywedge^? W \right\} = \langle \text{Pr} \rangle_{\text{nf}}.$$

▶ **Definition 25.** Let Pr be a problem such that $\langle \text{Pr} \rangle_{\text{nf}} = \text{Pr}'$. We say that $\langle \text{Pr} \rangle_{\text{nf}}$ is *reduced* when it consists of reduced constraints, and *successful* when $\text{Pr}' = \emptyset$ or contains only consistent reduced fixed-point constraints; otherwise, $\langle \text{Pr} \rangle_{\text{nf}}$ *fails*.

▶ **Definition 26.** A *solution* for a problem Pr is a pair of the form $\langle \Phi, \sigma \rangle$ where the following conditions are satisfied:

1. $\Phi \vdash \pi \curlywedge t\sigma$, if $\pi \curlywedge^? t \in \text{Pr}$;

2. $\Phi \vdash s\sigma \overset{\curlywedge^?}{\approx}_\alpha t\sigma$, if $s \overset{\curlywedge^?}{\approx}_\alpha t \in \text{Pr}$.

3. $X\sigma = X\sigma\sigma$ for all $X \in \text{Var}(\text{Pr})$ (the substitution is idempotent).

The *solution set* for a problem $\mathtt{Pr}$ is denoted by $\mathcal{U}(\mathtt{Pr})$.

The simplification rules (Table 1) specify a *unification algorithm*: we apply the simplification rules in a problem $\mathtt{Pr}$ until we reach a normal form $\langle\mathtt{Pr}\rangle_{\mathtt{nf}}$. In the case $\langle\mathtt{Pr}\rangle_{\mathtt{nf}}$ *fails* or contains reduced equational constraints, we say that $\mathtt{Pr}$ is *unsolvable*; otherwise, $\langle\mathtt{Pr}\rangle_{\mathtt{nf}}$ is *solvable* and its solution consists of the composition σ of substitutions applied through the simplification steps and the fixed-point context $\Phi = \{\pi \mathbin{⅄} X \mid \pi \mathbin{⅄}^{?} X \in \langle\mathtt{Pr}\rangle_{\mathtt{nf}}\}$.

▶ **Example 27** (Continuing example 24)**.** Notice that $\langle\Psi, \sigma\rangle$, where $\Psi = \{(a\ c_1) \mathbin{⅄} W, (c_2\ c_1) \mathbin{⅄} W\}$ and $\sigma = \{Y \mapsto b, X \mapsto (a\ b) \circ (b\ c) \cdot W\}$, is a solution for $\mathtt{Pr}$.

▶ **Theorem 28** (Correctness)**.** *Let* $\mathtt{Pr}$ *be a unification problem and* $\langle\mathtt{Pr}\rangle_{\mathtt{nf}} = \mathtt{Pr}'$, *then*
1. $\mathcal{U}(\mathtt{Pr}) = \mathcal{U}(\mathtt{Pr}')$, *and*
2. *if* $\mathtt{Pr}'$ *contains equational or inconsistent reduced fixed-point constraints then* $\mathcal{U}(\mathtt{Pr}) = \emptyset$.

Proof. The proof is by induction on the length of the derivation $\mathtt{Pr} \overset{n}{\Longrightarrow} \mathtt{Pr}'$. ◀

Base Case. $n = 0$. Then $\mathtt{Pr} = \mathtt{Pr}'$ and the result is trivial.

Induction Step. Suppose, $n > 0$ and consider the reduction chain

$$\mathtt{Pr} = \mathtt{Pr}_1 \Longrightarrow \ldots \Longrightarrow \mathtt{Pr}_{n-1} \Longrightarrow \mathtt{Pr}_n = \mathtt{Pr}'.$$

The proof follows by case analysis on the last rule applied in $\mathtt{Pr}_{n-1}$.
1. The rule is $(⅄at)$. In this case, $\mathtt{Pr}_{n-1} = \mathtt{Pr}'_{n-1} \uplus \{\pi \mathbin{⅄}^{?} a\} \Longrightarrow \mathtt{Pr}'_{n-1} = \mathtt{Pr}_n$, and $\pi(a) = a$. Let $\langle\Psi, \sigma\rangle \in \mathcal{U}(\mathtt{Pr}_{n-1})$, then
   a. $\Psi \vdash \pi' \mathbin{⅄} t\sigma$, for all $\pi' \mathbin{⅄}^{?} t \in \mathtt{Pr}'_{n-1}$
   b. $\Psi \vdash t\sigma \overset{⅄}{\approx}_{\alpha}^{?} s\sigma$, for all $t \overset{⅄}{\approx}_{\alpha}^{?} s \in \mathtt{Pr}'_{n-1}$;
   c. $X\sigma = X\sigma\sigma$, for all $X \in \mathtt{Var}(\mathtt{Pr}'_{n-1})$.
   Therefore, $\langle\Psi, \sigma\rangle \in \mathcal{U}(\mathtt{Pr}_n)$ and $\mathcal{U}(\mathtt{Pr}_{n-1}) \subseteq \mathcal{U}(\mathtt{Pr}_n)$. The other inclusion is trivial.
2. The rule is $(⅄var)$. In this case, $\mathtt{Pr}_{n-1} = \mathtt{Pr}'_{n-1} \uplus \{\pi \mathbin{⅄}^{?} \pi' \cdot X\} \Longrightarrow \mathtt{Pr}'_{n-1} \cup \{\pi^{(\pi')^{-1}} \mathbin{⅄}^{?} X\} = \mathtt{Pr}_n$, and $\pi' \neq Id$.
   Let $\langle\Psi, \sigma\rangle \in \mathcal{U}(\mathtt{Pr}_{n-1})$, then
   a. $\Psi \vdash \pi' \mathbin{⅄} t\sigma$, for all $\pi' \mathbin{⅄}^{?} t \in \mathtt{Pr}'_{n-1}$, and $\Psi \vdash \pi \mathbin{⅄} \pi' \cdot X\sigma$.
   b. $\Psi \vdash t\sigma \overset{⅄}{\approx}_{\alpha}^{?} s\sigma$, for all $t \overset{⅄}{\approx}_{\alpha}^{?} s \in \mathtt{Pr}'_{n-1}$;
   c. $X\sigma = X\sigma\sigma$, for all $X \in \mathtt{Var}(\mathtt{Pr}'_{n-1})$.
   Notice that

   $$\Psi \vdash \pi \mathbin{⅄} \pi' \cdot X\sigma \ \Rightarrow \Psi \vdash \pi \cdot (\pi' \cdot X\sigma) \overset{⅄}{\approx}_{\alpha} (\pi' \cdot X\sigma), \text{ hence}$$
   $$\Psi \vdash (\pi')^{-1} \circ \pi \circ \pi' \cdot (X\sigma) \overset{⅄}{\approx}_{\alpha} X\sigma \text{ via Lemma 12}$$
   $$\Rightarrow \Psi \vdash \pi^{(\pi')^{-1}} \mathbin{⅄} X\sigma.$$

   Therefore, $\langle\Psi, \sigma\rangle \in \mathcal{U}(\mathtt{Pr}_n)$ and $\mathcal{U}(\mathtt{Pr}_{n-1}) \subseteq \mathcal{U}(\mathtt{Pr}_n)$. The other inclusion is similar.
3. The rule is $(⅄\mathbf{abs})$. Then

   $$\mathtt{Pr}_{n-1} = \mathtt{Pr}' \uplus \{\pi \mathbin{⅄}^{?} [a]s\} \Longrightarrow \mathtt{Pr}' \cup \{(c_1\ c_2) \mathbin{⅄}^{?} \mathtt{Var}(s), \pi \mathbin{⅄}^{?} (a\ c_1).s\} = \mathtt{Pr}_n.$$

   where c_1 and c_2 are new names not occurring anywhere in the problem.
   Let $\langle\Psi, \sigma\rangle \in \mathcal{U}(\mathtt{Pr}_{n-1})$ be a solution for $\mathtt{Pr}_{n-1}$:
   a. $\Psi \vdash \pi' \mathbin{⅄} t\sigma$, for all $\pi' \mathbin{⅄}^{?} t \in \mathtt{Pr}'$ and $\Psi \vdash \pi \mathbin{⅄} ([a]s)\sigma$.
   b. $\Psi \vdash t\sigma \overset{⅄}{\approx}_{\alpha}^{?} s\sigma$, for all $t \overset{⅄}{\approx}_{\alpha}^{?} s \in \mathtt{Pr}'$.

Since $\Psi \vdash \pi \curlywedge ([a]s)\sigma$ and $([a]s)\sigma = [a]s\sigma$, it follows that $\Psi \vdash \pi \curlywedge ([a]s)\sigma$. From inversion and rule $(\curlywedge[\mathbf{a}])$, this implies that there exists a proof for $\Psi, (c_1\ c_2) \curlywedge \mathtt{Var}(s\sigma) \vdash \pi \curlywedge (a\ c_1).s\sigma$. Notice that we can always choose c_1 and c_2 such that $\underline{\mathtt{supp}((c_1\ c_2))} \cap \mathtt{supp}(s\sigma) = \emptyset$, from Lemma 11, it follows that $\Psi \vdash (c_1\ c_2) \curlywedge s\sigma$. Since $\Psi, \overline{(c_1\ c_2) \curlywedge \mathtt{Var}(s\sigma)} \vdash \pi \curlywedge (a\ c_1).s\sigma$, it follows that $\Psi \vdash \pi \curlywedge (a\ c_1).s\sigma$, by Proposition 16. ◀

▶ **Remark.** Theorem 18 guarantees the equivalence between $\approx_\alpha$ and $\overset{\curlywedge}{\approx}_\alpha$, therefore, we can associate the unification algorithm proposed, with the standard nominal unification algorithm proposed in [16]. The problem $\mathtt{Pr}$ introduced in Example 24, is equivalent to the nominal unification problem $\mathcal{P} = \{[a]\mathsf{f}(X,a) \approx_\alpha^? [b]\mathsf{f}((b\ c) \cdot W, (a\ c) \cdot Y\}$, and using the standard simplification rules [16]:

$$\mathcal{P} \overset{*}{\Longrightarrow} \overset{[Y \mapsto b]}{\Longrightarrow} \overset{*}{\Longrightarrow} \mathcal{P}' = \{X \approx_\alpha^? (a\ b) \cdot ((b\ c) \cdot W), a\#^? W\}$$
$$\overset{[X \mapsto (a\ b) \circ (b\ c) \cdot W]}{\Longrightarrow} \{a\#^? W\} = \mathcal{P}' \tag{5}$$

The pair $\langle \mathcal{P} \rangle_{\mathtt{sol}} = \langle \{a\#W\}, \delta \rangle$, where $\delta = \{Y/b, X \mapsto (a\ b) \circ (b\ c) \cdot W\}$ is a solution for $\mathcal{P}$. Using the translation $[\_]_\curlywedge$, we obtain $[\langle \mathcal{P} \rangle_{\mathtt{sol}}]_\curlywedge = \langle \{[a\#W]_\curlywedge\}, \delta \rangle = \langle (a\ c_a) \curlywedge W, \delta \rangle$, where c_a is a new name, which is equivalent to $\langle (a\ c_a) \curlywedge W, (c_a\ c_1) \curlywedge W, \delta \rangle$, for c_a and c_1 not occurring anywhere in $\mathcal{P}$. Therefore, $[\langle \mathcal{P}_{\mathtt{sol}} \rangle]_\curlywedge$ is a solution for $\mathtt{Pr} = \{[a]\mathsf{f}(X,a) \overset{\curlywedge}{\approx}_\alpha^?$ $[b]\mathsf{f}((b\ c) \cdot W, (a\ c) \cdot Y\}$. Similarly, from the solution $\langle \Psi, \sigma \rangle$ proposed in Example 27, we obtain $\langle [\Psi]_\#, \sigma \rangle = \langle a\#W, c_1\#W, c_2\#W, \sigma \rangle$, which is a solution for $\mathcal{P}$.

In the theorem below $\mathtt{Pr}_\curlywedge$ denotes a unification problem w.r.t. $\overset{\curlywedge}{\approx}_\alpha$ and $\curlywedge$, and $\mathcal{P}_\#$ denotes a unification problem w.r.t. $\approx_\alpha$ and $\#$.

▶ **Theorem 29.** *Let* $\mathtt{Pr}_\curlywedge$ *and* $\mathcal{P}_\#$ *be unification problems such that* $[\mathtt{Pr}_\curlywedge]^\# = \mathcal{P}_\#$ *and* $\langle \Psi, \sigma \rangle \in \mathcal{U}(\mathtt{Pr}_\curlywedge)$ *and* $\langle \Delta, \delta \rangle \in \mathcal{U}(\mathcal{P}_\#)$ *be solutions for* $\mathtt{Pr}_\curlywedge$ *and* $\mathcal{P}_\#$, *respectively. Then*
1. $\langle [\Psi]_\#, \sigma \rangle \in \mathcal{U}(\mathcal{P}_\#)$.
2. $\langle [\Delta]_\curlywedge, \delta \rangle \in \mathcal{U}(\mathtt{Pr}_\curlywedge)$.

5 Nominal C-unification via fixed-point constraints

In this section we propose an approach to nominal unification modulo commutativity via the notion of fixed-point constraints.

For example, assuming $+$ is commutative, i.e., $X + Y = Y + X$, a problem of the form

$$+\langle (a\ b) \cdot X, a \rangle \overset{\curlywedge}{\approx}_\alpha^? +\langle Y, X \rangle \tag{6}$$

can be solved by unifying $(a\ b) \cdot X$ with Y and a with X, or $(a\ b) \cdot X$ with X and a with Y.

In [2], a simplification algorithm for solving nominal C-unification was proposed. This algorithm was based on the standard nominal unification algorithm [16] where α-equivalence is defined w.r.t. the notion of freshness. Upon the input of a unification problem $\mathcal{P}$, the algorithm outputs a finite family of triples of the form $\langle \nabla, \sigma, P \rangle$, where ∇ is a freshness context, σ a substitution and P is a set of fixed-point constraints. In [3] we proved that even a simple unification problem, as $(a\ b) \cdot X \approx_\alpha X$ could produce an infinite and independent set of solutions, whenever the signature contains commutative function symbols: $\{X/a + b, X/f(a+b), X/[e]\langle a+b, b+a \rangle, \ldots\}$. Therefore, we could not provide a finite set of solutions consisting only of freshness constraints and substitutions. However, we remark that the problem $+\langle (a\ b) \cdot X, a \rangle \overset{\curlywedge}{\approx}_\alpha^? \langle Y, X \rangle$ mentioned above has in fact a finite number of most general solutions (indeed, two) if we solve it using fixed-point constraints. The most general unifiers are $\{X \mapsto a, Y \mapsto b\}$ and $\{Y \mapsto a, (a\ b) \curlywedge X\}$.

$$\frac{\pi(a) = a}{\Upsilon \vdash \pi \curlywedge_C a} \, (\curlywedge_{\mathbf{C}}\mathbf{a}) \qquad\qquad \frac{\mathrm{supp}(\pi^{\pi'^{-1}}) \subseteq \mathrm{supp}(\mathrm{perm}(\Upsilon|_X))}{\Upsilon \vdash \pi \curlywedge_C \pi' \cdot X} \, (\curlywedge_{\mathbf{C}}\mathbf{var})$$

$$\frac{\Upsilon \vdash \pi \curlywedge_C t}{\Upsilon \vdash \pi \curlywedge_C \mathsf{f}t} \, \mathsf{f} \neq + \, (\curlywedge_{\mathbf{C}}\mathbf{f}) \qquad \frac{\Upsilon \vdash \pi \cdot t_0 \stackrel{\curlywedge}{\approx}_{\alpha,C} t_i \quad \Upsilon \vdash \pi \cdot t_1 \stackrel{\curlywedge}{\approx}_{\alpha,C} t_{(i+1) \bmod 2}}{\Upsilon \vdash \pi \curlywedge_C +(t_0, t_1)} \, i = 0, 1 (\curlywedge_{\mathbf{C}}+)$$

$$\frac{\Upsilon \vdash \pi \curlywedge_C t_1 \quad \ldots \quad \Upsilon \vdash \pi \curlywedge_C t_n}{\Upsilon \vdash \pi \curlywedge_C (t_1, \ldots, t_n)} \, (\curlywedge_{\mathbf{C}}\mathbf{tuple}) \qquad \frac{\Upsilon, (c_1 \ c_2) \curlywedge_C \mathrm{Var}(t) \vdash \pi \curlywedge_C (a \ c_1) \cdot t}{\Upsilon \vdash \pi \curlywedge_C [a]t} \, (\curlywedge_{\mathbf{C}}\mathbf{abs})$$

Figure 5 Fixed-point rules modulo commutativity.

$$\frac{}{\Upsilon \vdash a \stackrel{\curlywedge}{\approx}_{\alpha,C} a} \, (\stackrel{\curlywedge}{\approx}_{\alpha,\mathbf{C}} \mathbf{a}) \qquad\qquad \frac{\Upsilon \vdash (\pi')^{-1} \circ \pi \curlywedge_C X}{\Upsilon \vdash \pi \cdot X \stackrel{\curlywedge}{\approx}_{\alpha,C} \pi' \cdot X} \, (\stackrel{\curlywedge}{\approx}_{\alpha,\mathbf{C}} \mathbf{var})$$

$$\frac{\Upsilon \vdash t \stackrel{\curlywedge}{\approx}_{\alpha,C} t'}{\Upsilon \vdash \mathsf{f}t \stackrel{\curlywedge}{\approx}_{\alpha,C} \mathsf{f}t'} \, (\stackrel{\curlywedge}{\approx}_{\alpha,\mathbf{C}} \mathbf{f}, \mathsf{f} \neq +) \qquad \frac{\Upsilon \vdash t_1 \stackrel{\curlywedge}{\approx}_{\alpha,C} t'_1 \quad \ldots \quad \Upsilon \vdash t_n \stackrel{\curlywedge}{\approx}_{\alpha,C} t'_n}{\Upsilon \vdash (t_1, \ldots, t_n) \stackrel{\curlywedge}{\approx}_{\alpha,C} (t'_1, \ldots, t'_n)} \, (\stackrel{\curlywedge}{\approx}_{\alpha,\mathbf{C}} \mathbf{tuple})$$

$$\frac{\Upsilon \vdash t \stackrel{\curlywedge}{\approx}_{\alpha,C} t'}{\Upsilon \vdash [a]t \stackrel{\curlywedge}{\approx}_{\alpha,C} [a]t'} \, (\stackrel{\curlywedge}{\approx}_{\alpha,\mathbf{C}} \mathbf{a}) \qquad \frac{\Upsilon \vdash s \stackrel{\curlywedge}{\approx}_{\alpha,C} (a \ b)t \quad \Upsilon, (c_1 \ c_2) \curlywedge_C \mathrm{Var}(t) \vdash (a \ c_1) \curlywedge_C t}{\Upsilon \vdash [a]s \stackrel{\curlywedge}{\approx}_{\alpha,C} [b]t} \, (\stackrel{\curlywedge}{\approx}_{\alpha,\mathbf{C}} \mathbf{ab})$$

$$\frac{\Upsilon \vdash s_0 \stackrel{\curlywedge}{\approx}_{\alpha,C} t_i \quad s_1 \stackrel{\curlywedge}{\approx}_{\alpha,C} t_{(i+1) \bmod 2}}{\Upsilon \vdash +\langle s_0, s_1 \rangle \stackrel{\curlywedge}{\approx}_{\alpha,C} +\langle t_0, t_1 \rangle} \, i = 0, 1 \, (\stackrel{\curlywedge}{\approx}_{\alpha,\mathbf{C}} +)$$

Figure 6 Rules for equality modulo commutativity.

▶ **Definition 30** (C-constraints). A C-*fixed-point constraint* is a pair of the form $\pi \curlywedge_C t$, of a permutation π and a term t. A C-α-*equality constraint* (for short, C-equality constraint) is a pair of the form $s \stackrel{\curlywedge}{\approx}_{\alpha,C} t$, for nominal terms s and t.

Intuitively, $s \stackrel{\curlywedge}{\approx}_{\alpha,C} t$ will mean that s and t are α-equivalent modulo commutativity of some function symbols, and $\pi \curlywedge_C t$ will mean that the permutation π has no effect on term t except for the commutativity of some subterms. For instance, $(a \ c) \curlywedge_C +\langle a, c \rangle$, but not $(a \ c) \curlywedge_C \mathsf{f}\langle a, c \rangle$, if f is not a commutative symbol

The notions of C-fixed-point contexts and C-judgements are defined as expected, and derivable according to the rules in Figures 5 and 6.

Rule ($\curlywedge_{\mathbf{C}}\mathbf{var}$) is similar to the previous one. Rule ($\stackrel{\curlywedge}{\approx}_{\alpha,\mathbf{C}} \mathbf{var}$) relies on the primitive notion of fixed-point constraints, it is equivalent to the rule given earlier. There is a branching rule ($\curlywedge_{\mathbf{C}}+$) for C-fixed-point constraints and a branching rule ($\stackrel{\curlywedge}{\approx}_{\alpha,\mathbf{C}} +$) for C-equality constraints (more precisely, in the case of C operators, there are two possible rules to apply, but we have written them in a compact way as one rule with parameter i). Technical results proven in Section 3 can be extended to C-constraints.

▶ **Theorem 31.** *Let* Υ, π *and* t *be a C-fixed-point context, a permutation and a nominal term, respectively.* $\Upsilon \vdash \pi \curlywedge_C t$ *iff* $\Upsilon \vdash \pi \cdot t \stackrel{\curlywedge}{\approx}_{\alpha,C} t$.

Proof. The proof is by induction on the structure of t, and follows the same lines of the proof of Theorem 18. ◀

$$\frac{\nabla \vdash s \approx_{\{\alpha,C\}} t}{\nabla \vdash f_k^E \, s \approx_{\{\alpha,C\}} f_k^E \, t} \,, \quad E \neq C \text{ or both } s \text{ and } t \text{ are not pairs} \quad (\approx_{\{\alpha,\mathbf{C}\}} \mathbf{app})$$

$$\frac{\nabla \vdash s_0 \approx_{\{\alpha,C\}} t_i, \ \ \nabla \vdash s_1 \approx_{\{\alpha,C\}} t_{(i+1)\,mod\,2}}{\nabla \vdash f_k^C \, \langle s_0, s_1 \rangle \approx_{\{\alpha,C\}} f_k^C \, \langle t_0, t_1 \rangle} \,, \quad i = 0, 1 \quad (\approx_{\{\alpha,\mathbf{C}\}} \mathbf{C})$$

Figure 7 Additional rules for $\{\alpha, C\}$-equivalence.

5.1 From freshness to C-fixed-point constraints

In [2] the relation $\approx_{\{\alpha,C\}}$ was defined as an extension of $\approx_\alpha$ (see the rules in Figures 3 and 4) with rules for commutative symbols:

Using the functions $[\_]_\curlywedge$ and $[\_]_\#$ defined in Section 3.2, we can obtain results that extend Lemma 19 and Theorem 20.

▶ **Lemma 32.** $\Delta \vdash a\#t \Rightarrow [\Delta]_\curlywedge^C, (c_1 \ c_2) \curlywedge_C \mathtt{Var}(t) \vdash (a \ c_1) \curlywedge_C t$,
where $[\Delta]_\curlywedge^C = \{\pi \curlywedge_C X \mid \pi \curlywedge X \in [\Delta]_\curlywedge\}$.

▶ **Theorem 33.**
1. $\Upsilon \vdash s \overset{\curlywedge}{\approx}_{\alpha,C} t \Rightarrow [\Upsilon]_\# \vdash s \approx_{\{\alpha,C\}} t$.
2. $\Delta \vdash s \approx_{\{\alpha,C\}} t \Rightarrow [\Delta]_\curlywedge^C \vdash s \overset{\curlywedge}{\approx}_{\alpha,C} t$.

5.2 Solving nominal C-unification problems via fixed-point constraints

Similarly to Section 4, we define the notion of nominal C-unification in terms of C-fixed-point constraints.

▶ **Definition 34.** A C-*unification problem* $\mathtt{Pr}$ consists of a finite set of C-equality and C-fixed-point constraints of the form $s \overset{\curlywedge?}{\approx}_C t$ and $\pi \curlywedge_C^? t$, respectively[3].

We write $\mathtt{Pr} \Longrightarrow_C \mathtt{Pr}'$ when $\mathtt{Pr}'$ is obtained from $\mathtt{Pr}$ by applying a simplification rule from Table 2 and we write $\overset{*}{\Longrightarrow}_C$ for the reflexive and transitive closure of $\Longrightarrow_C$. We omit the subindex when it is clear from the context.

▶ **Lemma 35.** *There is no infinite chain of reductions* $\Longrightarrow_C$ *starting from a C-unification problem* $\mathtt{Pr}$.

The simplification rules (Table 2) specify a C-*unification algorithm*: we apply the simplification rules in a problem $\mathtt{Pr}$ until we reach a normal form $\langle \mathtt{Pr} \rangle_{\mathtt{nf}}$. The notions of *solution, consistency, failure, correctness*, etc. obtained in Section 4 can be extended to C-unification.

▶ **Remark.** As with standard nominal unification, one can use the functions $[\_]_\#$ and $[\_]_\curlywedge$ to represent solutions $\langle \nabla, \sigma, P \rangle$ of nominal C-unification problems w.r.t. freshness constraints [2, 3] (where P is a set of fixed-point equations of the form $\pi.X \approx_{\{\alpha,C\}}^? X$) as solutions $\langle [\nabla]_\curlywedge \cup \{P_{\curlywedge_C}\}, \sigma \rangle$ of nominal C-unification problems via C-fixed-point constraints, where $P_{\curlywedge_C} = \{\pi \curlywedge_C X \mid \pi.X \approx_{\{\alpha,C\}}^? X \in P\}$.

---

[3] To ease the notation, we will denote $s \overset{\curlywedge?}{\approx}_C t$ by $s \approx^? t$.

Table 2 Simplification Rules for C-unification problems. In rules $(\curlywedge_C abs)$ and $(\overset{\curlywedge}{\approx}_{\alpha,C} abs2)$, c_1 and c_2 are new names.

$(\curlywedge_C at)$	$\mathtt{Pr} \uplus \{\pi \curlywedge_C^? a\}$	$\Longrightarrow$	$\mathtt{Pr}$, if $\pi(a) = a$
$(\curlywedge_C f)$	$\mathtt{Pr} \uplus \{\pi \curlywedge_C^? ft\}$	$\Longrightarrow$	$\mathtt{Pr} \cup \{\pi \curlywedge_C^? t\}, \mathsf{f} \neq +$
$(\curlywedge_C + 1)$	$\mathtt{Pr} \uplus \{\pi \curlywedge_C^? +\langle t_0, t_1\rangle\}$	$\Longrightarrow$	$\mathtt{Pr} \cup \{\pi \cdot t_0 \approx^? t_0, \pi \cdot t_1 \approx^? t_1\}$
$(\curlywedge_C + 2)$	$\mathtt{Pr} \uplus \{\pi \curlywedge_C^? +\langle t_0, t_1\rangle\}$	$\Longrightarrow$	$\mathtt{Pr} \cup \{\pi \cdot t_0 \approx^? t_1, \pi \cdot t_1 \approx^? t_0\}$
$(\curlywedge_C tuple)$	$\mathtt{Pr} \uplus \{\pi \curlywedge_C^? (\widetilde{t})_n\}$	$\Longrightarrow$	$\mathtt{Pr} \cup \{\pi \curlywedge_C^? t_1, \ldots, \pi \curlywedge_C^? t_n\}$
$(\curlywedge_C abs)$	$\mathtt{Pr} \uplus \{\pi \curlywedge_C^? [a]t\}$	$\Longrightarrow$	$\mathtt{Pr} \cup \{\pi \curlywedge_C^? (a\ c_1)\cdot t, \overline{(c_1\ c_2) \curlywedge_C^? \mathtt{Var}(t)}\}$
$(\curlywedge_C var)$	$\mathtt{Pr} \uplus \{\pi \curlywedge_C^? \pi' \cdot X\}$	$\Longrightarrow$	$\mathtt{Pr} \cup \{\pi^{(\pi')^{-1}} \curlywedge_C^? X\}$, if $\pi' \neq Id$
$(\overset{\curlywedge}{\approx}_{\alpha,C} a)$	$\mathtt{Pr} \uplus \{a \approx^? a\}$	$\Longrightarrow$	$\mathtt{Pr}$
$(\overset{\curlywedge}{\approx}_{\alpha,C} f)$	$\mathtt{Pr} \uplus \{ft \approx^? ft'\}$	$\Longrightarrow$	$\mathtt{Pr} \cup \{t \approx^? t'\}, \mathsf{f} \neq +$
$(\overset{\curlywedge}{\approx}_{\alpha,C} +1)$	$\mathtt{Pr} \uplus \{+\langle t_0, t_1\rangle \approx^? +\langle s_0, s_1\rangle\}$	$\Longrightarrow$	$\mathtt{Pr} \cup \{t_0 \approx^? s_0, t_1 \approx^? s_1\}$
$(\overset{\curlywedge}{\approx}_{\alpha,C} +2)$	$\mathtt{Pr} \uplus \{+\langle t_0, t_1\rangle \approx^? +\langle s_0, s_1\rangle\}$	$\Longrightarrow$	$\mathtt{Pr} \cup \{t_0 \approx^? s_1, t_1 \approx^? s_0\}$
$(\overset{\curlywedge}{\approx}_{\alpha,C} t)$	$\mathtt{Pr} \uplus \{(\widetilde{t})_n \approx^? (\widetilde{t'})_n\}$	$\Longrightarrow$	$\mathtt{Pr} \cup \{t_1 \approx^? t'_1, \ldots, t_n \approx^? t'_n\}$
$(\overset{\curlywedge}{\approx}_{\alpha,C} abs1)$	$\mathtt{Pr} \uplus \{[a]t \approx^? [a]t'\}$	$\Longrightarrow$	$\mathtt{Pr} \cup \{t \approx^? t'\}$
$(\overset{\curlywedge}{\approx}_{\alpha,C} abs2)$	$\mathtt{Pr} \uplus \{[a]t \approx^? [b]s\}$	$\Longrightarrow$	$\mathtt{Pr} \cup \{t \approx^? (a\ b)\cdot s, (a\ c_1) \curlywedge_C^? s,$
			$\overline{(c_1\ c_2) \curlywedge_C^? \mathtt{Var}(s)}\}$
$(\overset{\curlywedge}{\approx}_{\alpha,C} var)$	$\mathtt{Pr} \uplus \{\pi \cdot X \approx^? \pi' \cdot X\}$	$\Longrightarrow$	$\mathtt{Pr} \cup \{(\pi')^{-1} \circ \pi \curlywedge_C^? X\}$
$(\overset{\curlywedge}{\approx}_{\alpha,C} inst1)$	$\mathtt{Pr} \uplus \{\pi \cdot X \approx^? t\}$	$\overset{[X \mapsto \pi^{-1}.t]}{\Longrightarrow}$	$\mathtt{Pr}\{X \mapsto \pi^{-1}.t\}$, if $X \notin \mathtt{Var}(t)$
$(\approx_\alpha inst2)$	$\mathtt{Pr} \uplus \{t \approx^? \pi \cdot X\}$	$\overset{[X \mapsto \pi^{-1}.t]}{\Longrightarrow}$	$\mathtt{Pr}\{X \mapsto \pi^{-1}.t\}$, if $X \notin \mathtt{Var}(t)$

6 Conclusions and Future Work

The notion of fixed-point constraints allowed us to obtain a finite representation of solutions for nominal C-unification problems. This brings a novel alternative to standard nominal unification approaches in which just the algebra of atom permutations and the logic of freshness constraints are used to implement equational reasoning (e.g., [1, 5, 6, 7, 9]), and in particular to their extensions modulo commutativity, for which only infinite representations were possible in the standard approach. With the new proposed approach the development of algorithms for the generation of solutions of nominal equational problems modulo theories such as C, AC, etc would be simplified avoiding with the use of fixed-point constraints the development of procedures for the generation of infinite independent sets of solutions.

In future work we plan to extend this approach to matching and unification modulo different equational theories as well as to the treatment of equational problems in nominal rewriting modulo.

References

1 T. Aoto and K. Kikuchi. *A Rule-Based Procedure for Equivariant Nominal Unification.* In *Pre-proc. of Higher-Order Rewriting (HOR)*, pages 1–5, 2016.

2 M. Ayala-Rincón, W. Carvalho-Segundo, M. Fernández, and D. Nantes-Sobrinho. *Nominal C-Unification.* In *Pre-proc. of the 27th Int. Symp. Logic-based Program Synthesis and Transformation (LOPSTR)*, pages 1–15, 2017. URL: https://arxiv.org/abs/1709.05384.

3 M. Ayala-Rincón, W. Carvalho-Segundo, M. Fernández, and D. Nantes-Sobrinho. *On Solving Nominal Fixpoint Equations.* In *Proc. of the 11th Int. Symp. on Frontiers of Combining*

  Systems (FroCoS), volume 10483 of *Lecture Notes in Computer Science*, pages 209–226. Springer, 2017. `doi:10.1007/978-3-319-66167-4_12`.

4 M. Ayala-Rincón, W. de Carvalho Segundo, M. Fernández, and D. Nantes-Sobrinho. A formalisation of nominal α-equivalence with A and AC function symbols. *Electronic Notes in Theoretical Computer Science*, 332:21–38, 2017. `doi:10.1016/j.entcs.2017.04.003`.

5 C. F. Calvès. Unifying Nominal Unification. In *24th International Conference on Rewriting Techniques and Applications (RTA 2013)*, volume 21 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 143–157, 2013. `doi:10.4230/LIPIcs.RTA.2013.143`.

6 C. F. Calvès and M. Fernández. A Polynomial Nominal Unification Algorithm. *Theoretical Computer Science*, 403(2-3):285–306, 2008. `doi:10.1016/j.tcs.2008.05.012`.

7 J. Cheney. *Equivariant unification*. *Journal of Automated Reasoning*, 45(3):267–300, 2010. `doi:10.1007/s10817-009-9164-3`.

8 M. Fernández and M. J. Gabbay. *Nominal Rewriting*. *Information and Computation*, 205(6):917–965, 2007. `doi:10.1016/j.ic.2006.12.002`.

9 M. Fernández, M. J. Gabbay, and I. Mackie. *Nominal Rewriting Systems*. In *Proc. of the 6th Int. Conf. on Principles and Practice of Declarative Programming (PPDP)*, pages 108–119. ACM Press, 2004. `doi:10.1145/1013963.1013978`.

10 M. J. Gabbay. *A Theory of Inductive Definitions With α-equivalence*. PhD thesis, DPMMS and Trinity College, University of Cambridge, 2000.

11 M. J. Gabbay and A. Mathijssen. Capture-avoiding substitution as a nominal algebra. *Formal Aspects of Computing*, 20(4-5):451–479, 2008. `doi:10.1007/s00165-007-0056-1`.

12 M. J. Gabbay and A. M. Pitts. *A New Approach to Abstract Syntax with Variable Binding*. *Formal Aspects of Computing*, 13(3-5):341–363, 2002. `doi:10.1007/s001650200016`.

13 A. M. Pitts. Nominal Logic, a First Order Theory of Names and Binding. *Information and Computation*, 186(2):165–193, 2003. `doi:10.1016/S0890-5401(03)00138-X`.

14 A. M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, 2013.

15 M. Schmidt-Schauß, T. Kutsia, J. Levy, and M. Villaret. *Nominal Unification of Higher Order Expressions with Recursive Let*. In *26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR), Revised Selected Papers*, volume 10184 of *Lecture Notes in Computer Science*, pages 328–344, 2016. `doi:10.1007/978-3-319-63139-4_19`.

16 C. Urban, A. M. Pitts, and M. J. Gabbay. *Nominal Unification*. *Theoretical Computer Science*, 323(1-3):473–497, 2004. `doi:10.1016/j.tcs.2004.06.016`.

Strict Ideal Completions of the Lambda Calculus

Patrick Bahr
IT University of Copenhagen, Denmark
paba@itu.dk
 https://orcid.org/0000-0003-1600-8261

Abstract

The infinitary lambda calculi pioneered by Kennaway et al. extend the basic lambda calculus by metric completion to infinite terms and reductions. Depending on the chosen metric, the resulting infinitary calculi exhibit different notions of *strictness*. To obtain infinitary normalisation and infinitary confluence properties for these calculi, Kennaway et al. extend β-reduction with infinitely many '⊥-rules', which contract *meaningless terms* directly to ⊥. Three of the resulting *Böhm reduction* calculi have unique infinitary normal forms corresponding to Böhm-like trees.

In this paper we develop a corresponding theory of infinitary lambda calculi based on ideal completion instead of metric completion. We show that each of our calculi conservatively extends the corresponding metric-based calculus. Three of our calculi are infinitarily normalising and confluent; their unique infinitary normal forms are exactly the Böhm-like trees of the corresponding metric-based calculi. Our calculi dispense with the infinitely many ⊥-rules of the metric-based calculi. The fully non-strict calculus (called 111) consists of only β-reduction, while the other two calculi (called 001 and 101) require two additional rules that precisely state their strictness properties: $\lambda x.\bot \to \bot$ (for 001) and $\bot\, M \to \bot$ (for 001 and 101).

1 Introduction

In their seminal work on infinitary lambda calculus, Kennaway et al. [10] study different infinitary variants of the lambda calculus, which are obtained by extending the ordinary lambda calculus by means of metric completion. Different variants of the calculus are obtained by choosing a different metric. The 'standard' metric on terms measures the distance between two terms depending on how deep one has to go into the term structure to distinguish two terms. For example the term $x\, y$ is closer to the term $x\, z$ than to the term x, because in the former case both terms are applications whereas in the latter case one term is an application and the other is a variable.

The different metric spaces arise by changing the way in which we measure depth. Kennaway et al. [10] indicate this using a binary triple abc with $a, b, c \in \{0, 1\}$, where $a = 0$ indicates that we do not count lambda abstractions when calculating the depth, and $b = 0$ or $c = 0$ indicates that we do not count the left or the right side of applications, respectively. More intuitively these three parameters can be interpreted as indicating *strictness*. For example, $a = 0$ indicates that lambda abstraction is strict, i.e. if M diverges, then so does $\lambda x.M$.

Since the set of infinite terms is constructed from the set of finite terms by means of metric completion, each calculus has a different universe of terms, as well as a different mode of convergence, which is based on the topology induced by the metric. For instance, from the lambda term $N = (\lambda x.x\,x\,y)(\lambda x.x\,x\,y)$, we can derive the infinite reduction $N \to N\,y \to N\,y\,y \to \dots$. In the fully non-strict calculus, where $abc = 111$, this reduction converges to the infinite term $M = \dots y\,y\,y$ (i.e. M satisfies $M = M\,y$). By contrast, in the calculus 101, which is strict on the left-hand side of every application, this reduction does not converge. In fact, M is not even a valid term in the 101 calculus.

In order to deal with divergence as exemplified for the 101 calculus above, Kennaway et al. [10] extend standard β-reduction to *Böhm reduction* by adding rules of the form $M \to \bot$, for each term M that causes divergence such as the term N in the 101 calculus. The resulting 001, 101, and 111 calculi based on Böhm reduction have unique normal forms, which correspond to the well-known *Böhm Trees* [14, 6], *Levy-Longo Trees* [13, 15] and *Berarducci Trees* [7], respectively.

In this paper, we introduce infinitary lambda calculi that are based on ideal completion instead of metric completion with the goal of directly dealing with diverging terms without the need for additional reduction rules that contract diverging terms immediately to $\bot$. To this end, we devise for each metric of the calculi of Kennaway et al. [10] a corresponding partial order with the following property: Ideal completion of the set of finite lambda terms yields the same set of infinite lambda terms as the corresponding metric completion (Section 3). We also find a strong correspondence between the modes of convergence induced by these structures: Each ideal completion yields a complete semilattice structure, which means that the *limit inferior* is always defined. We show that this limit inferior is a conservative extension of the limit in the corresponding metric completion in the sense that both modes of convergence coincide on total lambda terms, i.e. terms without $\bot$ (Section 3).

Based on these partial order structures we define infinitary lambda calculi by a straightforward instantiation of transfinite abstract reduction systems [2]. We find that the ideal completion calculi form a conservative extension of the metric completion calculi of Kennaway et al. [10] (Section 4). Moreover, in analogy to Blom [9] and Bahr [3], we find that the differences between the ideal completion approach and the metric completion approach are compensated for by adding $\bot$-rules to the metric calculi in the style of Kennaway et al. [11] (Section 5). Finally, we also show infinitary normalisation for our ideal completion calculi and infinitary confluence for the 001, 101, and 111 calculi (Section 5). However, in order to obtain infinitary confluence for 001 and 101, we need to extend β-reduction with two additional rules that precisely capture the strictness properties of these calculi: $\lambda x.\bot \to \bot$ (for 001) and $\bot M \to \bot$ (for 001 and 101). In Section 6, we give a brief overview of related work.

2 The Metric Completion

In this section, we introduce infinite lambda terms as the result of metric completion of the set of finite lambda terms. Before we get started, we introduce some basic notions about transfinite sequences and lambda terms. We presume basic familiarity with metric spaces and ordinal numbers.

A *sequence* over a set A of length α is a mapping from an ordinal α into A and is written as $(a_\iota)_{\iota<\alpha}$, which indicates the mapping $\iota \mapsto a_\iota$; the notation $|(a_\iota)_{\iota<\alpha}|$ denotes the length α of $(a_\iota)_{\iota<\alpha}$. If α is a limit ordinal, then $(a_\iota)_{\iota<\alpha}$ is called *open*; otherwise it is called *closed*. If $(a_\iota)_{\iota<\alpha}$ is finite, it is also written as $\langle a_0, \dots, a_{\alpha-1}\rangle$; in particular, $\langle\rangle$ denotes the empty

sequence. We write $S \cdot T$ for the *concatenation* of two sequences S and T; S is called a (*proper*) *prefix* of T, denoted $S \leq T$ (resp. $S < T$) if there is a (non-empty) sequence S' such that $S \cdot S' = T$. The unique prefix of a sequence S of length $\beta \leq |S|$ is denoted by $S|_\beta$.

We consider lambda terms with an additional symbol $\bot$; the resulting set of *lambda terms* $\Lambda_\bot$ is inductively defined by the following grammar:

$$M, N ::= \bot \mid x \mid \lambda x.M \mid MN$$

where x is drawn from a countably infinite set $\mathcal{V}$ of variable symbols. The set of *total lambda terms* Λ is the subset of lambda terms in $\Lambda_\bot$ that do not contain $\bot$. Occurrences of a variable x in a subterm $\lambda x.M$ are called *bound*; other occurrences are called *free*. We use the notation $M[x \to y]$ to replace all free occurrences of the variable x in M with the variable y. We use finite sequences over $\{0, 1, 2\}$, called *positions*, to point to subterms of a lambda term; we write $\mathcal{P}$ for the set of all positions. For each $M \in \Lambda_\bot$, $\mathcal{P}(M)$ denotes the set of positions of M (excluding '$\bot$'s) recursively defined as follows: $\mathcal{P}(\bot) = \emptyset$, $\mathcal{P}(x) = \{\langle\rangle\}$, $\mathcal{P}(M_1 \, M_2) = \{\langle\rangle\} \cup \{\langle i\rangle \cdot p \mid i \in \{1, 2\}, p \in \mathcal{P}(M_i)\}$, and $\mathcal{P}(\lambda x.M) = \{\langle\rangle\} \cup \{\langle 0\rangle \cdot p \mid p \in \mathcal{P}(M)\}$.

A *conflict* [10] between two lambda terms M, N is a position $p \in \mathcal{P}(M) \cup \mathcal{P}(N)$ such that: (a) if $p = \langle\rangle$, then M and N are not identical variables, not both $\bot$, not both applications, and not both abstractions; (b) if $p = \langle i\rangle \cdot q$ and $i \in \{1, 2\}$, then $M = M_1 M_2$, $N = N_1 N_2$, and q is a conflict of M_i and N_i; (c) if $p = \langle 0\rangle \cdot q$, then $M = \lambda x.M'$, $N = \lambda y.N'$, and q is a conflict of $M'[x \to z]$ and $N'[y \to z]$, where z is a fresh variable occurring neither in M nor N. The terms M and N are said to be *α-equivalent* if they have no conflicts. By convention we identify α-equivalent terms (i.e. $\Lambda_\bot$ and Λ are assumed to be quotients by α-equivalence).

▶ **Definition 2.1.** Given a triple $\overline{a} = a_0 a_1 a_2 \in \{0, 1\}^3$, called *strictness signature*, a position is called *$\overline{a}$-strict* if it is of the form $q \cdot \langle i\rangle$ with $a_i = 0$; otherwise it is called *$\overline{a}$-non-strict*. If $\overline{a}$ is clear from the context, we only say *strict* resp. *non-strict*.

That is, a strictness signature indicates strictness by 0 and non-strictness by 1. For example, if $\overline{a} = 011$, lambda abstraction is strict, and application is non-strict both from the left and the right. We shall see what this means shortly: Following Kennaway et al. [10], we derive, from a strictness signature $\overline{a}$, a depth measure $|\cdot|^{\overline{a}}$, which counts the number of non-strict, non-empty prefixes of a position. From this depth measure we then derive a corresponding metric $\mathbf{d}^{\overline{a}}$ on lambda terms.

▶ **Definition 2.2.** Given a strictness signature $\overline{a}$, the *$\overline{a}$-depth* of a position p, denoted $|p|^{\overline{a}}$, is recursively defined as $|\langle\rangle|^{\overline{a}} = 0$ and $|q \cdot \langle i\rangle|^{\overline{a}} = |q|^{\overline{a}} + a_i$. The *$\overline{a}$-distance* $\mathbf{d}^{\overline{a}}(M, N)$ between two terms $M, N \in \Lambda_\bot$ is 0 if M and N are α-equivalent and otherwise 2^{-d}, where d is the least number satisfying $d = |p|^{\overline{a}}$ for some conflict p of M and N.

Kennaway et al. [10] showed that the pair $(\Lambda_\bot, \mathbf{d}^{\overline{a}})$ forms an ultrametric space for any $\overline{a}$. Intuitively, the consequence of the definition of these metric spaces is that sequences of terms, such as the sequence $N, N\,y, N\,y\,y, \ldots$, only converge if conflicts between consecutive terms are guarded by an increasing number of non-strict positions. In the example, conflicts between consecutive terms are guarded by an increasing stack of applications to y. If $a_1 = 1$, these applications correspond to non-strict positions, and thus the sequence converges. However, if $a_1 = 0$, the sequence does not converge.

We turn now to the metric completion. To facilitate later definitions and to illustrate the resulting structures, we use a partial function representation in the form of lambda trees taken

from Blom [9], which will serve as mediator between metric completion and ideal completion.[1]
A lambda tree is a (possibly infinite) labelled tree where a label λ indicates abstraction and
@ indicates application; labels in $\mathcal{V}$ indicate free variables and a label $p \in \mathcal{P}$ indicates a
variable that is bound by an abstraction at position p. There is no label corresponding to $\bot$,
which instead is represented as a 'hole' in the tree. We write $\mathcal{D}(f)$ to denote the domain of a
partial function f, and $f(p) \simeq g(q)$ to indicate that the partial functions f and g are either
both undefined or have the same value at p and q, respectively.

▶ **Definition 2.3.** A *lambda tree* is a partial function $t \colon \mathcal{P} \rightharpoonup \mathcal{L}$ with $\mathcal{L} = \{\lambda, @\} \uplus \mathcal{P} \uplus \mathcal{V}$ so
that

  (a) $p \cdot \langle 0 \rangle \in \mathcal{D}(t)$ $\implies$ $t(p) = \lambda,$
  (b) $p \cdot \langle 1 \rangle \in \mathcal{D}(t)$ or $p \cdot \langle 2 \rangle \in \mathcal{D}(t)$ $\implies$ $t(p) = @,$ and
  (c) $t(p) = q$, where $q \in \mathcal{P}$ $\implies$ $q \leq p$ and $t(q) = \lambda.$

As one would expect, the domain $\mathcal{D}(t)$ of a lambda tree t is prefix closed.

   The set of all lambda trees is denoted $\mathcal{T}_\bot^\infty$. The set of $\bot$-*positions* in t, denoted $\mathcal{D}_\bot(t)$,
is the smallest set satisfying (a) $\langle \rangle \notin \mathcal{D}(t)$ implies $\langle \rangle \in \mathcal{D}_\bot(t)$; (b) $t(p) = \lambda, p \cdot \langle 0 \rangle \notin \mathcal{D}(t)$
implies $p \cdot \langle 0 \rangle \in \mathcal{D}_\bot(t)$; and (c) $t(p) = @, p \cdot \langle i \rangle \notin \mathcal{D}(t), i \in \{1, 2\}$ implies $p \cdot \langle i \rangle \in \mathcal{D}_\bot(t)$. A
lambda tree t is called *total* if $\mathcal{D}_\bot(t)$ is empty. The set of all total lambda trees is denoted
$\mathcal{T}^\infty$. A lambda tree t is called *finite* if $\mathcal{D}(t)$ is a finite set. The set of all finite (total) lambda
trees is denoted $\mathcal{T}_\bot$ (respectively $\mathcal{T}$). A *renaming* of a lambda tree t is a lambda tree s such
that there is a bijection $f \colon \mathcal{V} \to \mathcal{V}$ with the following properties: $s(p) = t(p)$ if $t(p) \in \mathcal{L} \setminus \mathcal{V}$,
$s(p) = f(t(p))$ if $t(p) \in \mathcal{V}$, and otherwise $s(p)$ is undefined.

In order to avoid confusion, we use upper case letters M, N for lambda terms and lower case
letters s, t, u for lambda trees. Below, we give a bijection from lambda terms to finite lambda
trees that should help illustrate the idea behind lambda trees. At the heart of this bijection
are the following constructions based on Blom [9]:

▶ **Definition 2.4.** Given lambda trees $t, t_1, t_2 \in \mathcal{T}_\bot^\infty$ and a variable $x \in \mathcal{V}$, let $\bot, \mathsf{x}, \lambda\mathsf{x}.t$ and
$t_1 \, t_2$ be partial functions of type $\mathcal{P} \rightharpoonup \mathcal{L}$ defined by their graph as follows:

$$\bot = \emptyset \qquad \mathsf{x} = \{(\langle\rangle, x)\}$$

$$\lambda\mathsf{x}.t = \{(\langle\rangle, \lambda)\} \cup \{(\langle 0 \rangle \cdot p, l) \mid l \in \{\lambda, @\} \uplus \mathcal{V} \setminus \{x\}, (p, l) \in t\}$$

$$\cup \{(\langle 0 \rangle \cdot p, \langle 0 \rangle \cdot q) \mid q \in \mathcal{P}, (p, q) \in t\} \cup \{(\langle 0 \rangle \cdot p, \langle\rangle) \mid (p, x) \in t\}$$

$$t_1 \, t_2 = \{(\langle\rangle, @)\} \cup \{(\langle i \rangle \cdot p, l) \mid i \in \{1, 2\}, l \in \{\lambda, @\} \uplus \mathcal{V}, (p, l) \in t_i\}$$

$$\cup \{(\langle i \rangle \cdot p, \langle i \rangle \cdot q) \mid i \in \{1, 2\}, q \in \mathcal{P}, (p, q) \in t_i\}$$

One can easily check that each of the above four constructions yields a lambda tree, where
$\bot$ is the empty lambda tree, x the lambda tree consisting of a single free variable x, $\lambda\mathsf{x}.t$ is a
lambda abstraction over x with body t, and $t_1 \, t_2$ is an application of t_1 to t_2. The following
translation of lambda terms to finite lambda trees illustrates the use of these constructions:

▶ **Definition 2.5.** Let $[\![\cdot]\!] : \Lambda_\bot \to \mathcal{T}_\bot$ be defined recursively as follows:

$$[\![\bot]\!] = \bot \qquad\qquad [\![\lambda x.M]\!] = \lambda\mathsf{x}. [\![M]\!] \qquad\qquad [\![x]\!] = \mathsf{x} \qquad\qquad [\![M \, N]\!] = [\![M]\!] \, [\![N]\!]$$

   One can easily check that $[\![\cdot]\!] : \Lambda_\bot \to \mathcal{T}_\bot$ is indeed a bijection, which, if restricted to Λ, is
a bijection from Λ to $\mathcal{T}$. Moreover, one can show that each $t \in \mathcal{T}_\bot^\infty$ with some $\langle i \rangle \cdot p \in \mathcal{D}(t)$

---

[1] In the companion report [5] we give a direct proof of the correspondence between metric and ideal
     completion based on the meta theory of Majster-Cederbaum and Baier [16].

is equal to $\lambda x.t'$ if $i = 0$ and to $t_1\,t_2$ if $i \in \{1, 2\}$, for some $t', t_1, t_2 \in \mathcal{T}_\bot^\infty$. Following this observation, we define, for each $t \in \mathcal{T}_\bot^\infty$ and $p \in \mathcal{D}(t)$, the *subtree* of t at p, denoted $t|_p$, by induction on p as follows: $t|_{\langle\rangle} = t$, $\lambda x.t|_{\langle 0\rangle \cdot p} = t|_p$, and $t_1\,t_2|_{\langle i\rangle \cdot p} = t_i|_p$ for $i \in \{1, 2\}$. One can easily check that $t|_p$ is uniquely defined modulo renaming of free variables.

▶ **Definition 2.6.** An *infinite branch* in a lambda tree $t \in \mathcal{T}_\bot^\infty$ is an infinite sequence S such that each proper prefix of S is in $\mathcal{D}(t)$. We call a proper prefix of S a *position along S*.

Note that by instantiating König's Lemma to lambda trees, we know that a lambda tree is infinite iff it has an infinite branch.

The idea of the metric $\mathbf{d}^{\overline{a}}$ on lambda terms is to disallow (in the ensuing metric completion) infinite branches that have only finitely many non-strict positions along them. The following definition makes this restriction explicit on lambda trees:

▶ **Definition 2.7.** An infinite branch S of a lambda tree t is called $\overline{a}$-*bounded* if the $\overline{a}$-depth of all positions along S is bounded by some $n < \omega$, i.e. $|p|^{\overline{a}} < n$ for all $p < S$. The lambda tree t is called $\overline{a}$-*unguarded* if it has an $\overline{a}$-bounded infinite branch S. Otherwise, t is called $\overline{a}$-*guarded*. The set of all $\overline{a}$-guarded (total) lambda trees is denoted $\mathcal{T}_\bot^{\overline{a}}$ (respectively $\mathcal{T}^{\overline{a}}$). In particular, $\mathcal{T}_\bot^{000} = \mathcal{T}_\bot$ and $\mathcal{T}_\bot^{111} = \mathcal{T}_\bot^\infty$.

For example, the lambda tree s with $s = s\,\mathsf{y}$ is 101-unguarded while t with $t = \lambda\mathsf{y}.t\,\mathsf{y}$ is 101-guarded as each application is guarded by an abstraction (which is non-strict).

For each strictness signature $\overline{a}$, we give a metric $\mathbf{d}_{\mathcal{T}}^{\overline{a}}$ on lambda trees that corresponds to the metric $\mathbf{d}^{\overline{a}}$ on lambda terms.

▶ **Definition 2.8.** For each two lambda trees $s, t \in \mathcal{T}_\bot^\infty$, define $\mathbf{d}_{\mathcal{T}}^{\overline{a}}(s, t) = 0$ if $s = t$ and otherwise $\mathbf{d}_{\mathcal{T}}^{\overline{a}}(s, t) = 2^{-d}$, where d is the least $|p|^{\overline{a}}$ with $s(p) \not\simeq t(p)$.

From the characterisation of the metric completion of $(\Lambda_\bot, \mathbf{d}^{\overline{a}})$ from Kennaway et al. [10, Lemma 7] we know that the metric space of $\overline{a}$-guarded lambda trees $(\mathcal{T}_\bot^{\overline{a}}, \mathbf{d}_{\mathcal{T}}^{\overline{a}})$ is indeed the metric completion of $(\Lambda_\bot, \mathbf{d}^{\overline{a}})$ with the isometric embedding $[\![\cdot]\!] : \Lambda_\bot \to \mathcal{T}_\bot$ (cf. the companion report [5]). Analogously, $(\mathcal{T}^{\overline{a}}, \mathbf{d}_{\mathcal{T}}^{\overline{a}})$ is the metric completion of $(\Lambda, \mathbf{d}^{\overline{a}})$.

3 The Ideal Completion

In this section, we present an alternative to the metric completion from Section 2 that is based on a family of partial orders on lambda terms indexed by strictness signatures. In the following we assume basic familiarity with order theory.

▶ **Definition 3.1.** Given a strictness signature $\overline{a}$, the partial order $\leq_\bot^{\overline{a}}$ is the least transitive, reflexive order on $\Lambda_\bot$ satisfying the following for all $M, M', N, N' \in \Lambda_\bot$ and $x \in \mathcal{V}$:

(a) $\bot \leq_\bot^{\overline{a}} M$
(b) $\lambda x.M \leq_\bot^{\overline{a}} \lambda x.M'$ if $M \leq_\bot^{\overline{a}} M'$ and $M \neq \bot$ or $a_0 = 1$
(c) $MN \leq_\bot^{\overline{a}} M'N$ if $M \leq_\bot^{\overline{a}} M'$ and $M \neq \bot$ or $a_1 = 1$
(d) $MN \leq_\bot^{\overline{a}} MN'$ if $N \leq_\bot^{\overline{a}} N'$ and $N \neq \bot$ or $a_2 = 1$

For the case that $\overline{a} = 111$, we obtain the partial order $\leq_\bot$ that is typically used for ideal completions. This order is fully monotone, i.e. $M \leq_\bot M'$ implies $\lambda x.M \leq_\bot \lambda x.M'$, $MN \leq_\bot M'N$ and $NM \leq_\bot NM'$. By contrast, $\leq_\bot^{\overline{a}}$ restricts monotonicity of abstraction in case $a_0 = 0$ and of application in case $a_1 = 0$ or $a_2 = 0$. Intuitively, we have $M \leq_\bot^{\overline{a}} N$ iff N can be obtained from M by replacing occurrences of $\bot$ in M at non-strict positions with

arbitrary terms. For example, if $\overline{a} = 001$, then neither $\lambda x.\bot \leq_\bot^{\overline{a}} \lambda x.x\,x$ nor $\lambda x.\bot\,x \leq_\bot^{\overline{a}} \lambda x.x\,x$; but we do have that $\lambda x.x\,\bot \leq_\bot^{\overline{a}} \lambda x.x\,x$.

With this intuition in mind, we translate $\leq_\bot^{\overline{a}}$ to a corresponding order $\trianglelefteq_\bot^{\overline{a}}$ on lambda trees as follows:

▶ **Definition 3.2.** Given lambda trees $s, t \in \mathcal{T}_\bot^\infty$, we have $s \trianglelefteq_\bot^{\overline{a}} t$ if

(a) $\mathcal{D}(s) \subseteq \mathcal{D}(t)$,

(b) $s(p) = t(p)$ for all $p \in \mathcal{D}(s)$, and

(c) $p \in \mathcal{D}(s) \implies p \cdot \langle i \rangle \in \mathcal{D}(s)$ for all $\overline{a}$-strict positions $p \cdot \langle i \rangle \in \mathcal{D}(t)$.

Conditions (a) and (b) alone would give us the corresponding order for the standard partial order $\leq_\bot$. Condition (c) ensures that the partial order $\trianglelefteq_\bot^{\overline{a}}$ may not fill a hole in a strict position in the left-hand side tree.

One can check that $(\mathcal{T}_\bot^\infty, \trianglelefteq_\bot^{\overline{a}})$ forms a partially ordered set. Moreover, we have the following correspondence between the two families of orders $\leq_\bot^{\overline{a}}$ and $\trianglelefteq_\bot^{\overline{a}}$:

▶ **Proposition 3.3.** $[\![\cdot]\!] : (\Lambda_\bot, \leq_\bot^{\overline{a}}) \to (\mathcal{T}_\bot, \trianglelefteq_\bot^{\overline{a}})$ *is an order isomorphism.*

For the remainder of this section, we turn our focus to the partial orders $\trianglelefteq_\bot^{\overline{a}}$ on lambda trees. In particular, we show that $(\mathcal{T}_\bot^{\overline{a}}, \trianglelefteq_\bot^{\overline{a}})$ forms a *complete semilattice* and that it is (order isomorphic to) the ideal completion of $(\Lambda_\bot, \leq_\bot^{\overline{a}})$. A complete semilattice is a partially ordered set $(A, \leq)$ that is a *complete partial order* (*cpo*) and that has a *greatest lower bound* (*glb*) $\bigsqcap B$ for every *non-empty* set $B \subseteq A$.[2] A partially ordered set $(A, \leq)$ is a cpo if it has a least element, and each directed set D in $(A, \leq)$ has a *least upper bound* (*lub*) $\bigsqcup D$; a set $D \subseteq A$ is called directed if for each two $a, b \in D$ there is some $c \in D$ with $a, b \leq c$.

In particular, for any sequence $(a_\iota)_{\iota < \alpha}$ in a complete semilattice, its *limit inferior*, defined by $\liminf_{\iota \to \alpha} a_\iota = \bigsqcup_{\beta < \alpha} \left(\bigsqcap_{\beta \leq \iota < \alpha} a_\iota \right)$, exists. While the metric completion lambda calculi are based on the limit of the underlying metric space, our ideal completion lambda calculi are based on the limit inferior.

To show that $(\mathcal{T}_\bot^{\overline{a}}, \trianglelefteq_\bot^{\overline{a}})$ forms a complete semilattice structure, we construct the appropriate lubs and glbs:

▶ **Theorem 3.4** (cpo $(\mathcal{T}_\bot^{\overline{a}}, \trianglelefteq_\bot^{\overline{a}})$). *The partially ordered set* $(\mathcal{T}_\bot^{\overline{a}}, \trianglelefteq_\bot^{\overline{a}})$ *forms a complete partial order. In particular, the lub t of a directed set D satisfies the following:*

$$\mathcal{D}(t) = \bigcup_{s \in D} \mathcal{D}(s) \qquad\qquad s(p) = t(p) \quad \text{for all } s \in D, p \in \mathcal{D}(s)$$

Proof sketch. The lambda tree $\bot$ is the least element in $(\mathcal{T}_\bot^{\overline{a}}, \trianglelefteq_\bot^{\overline{a}})$. Construct the lub t of D as follows: $t(p) = s(p)$ iff there is some $s \in D$ with $p \in \mathcal{D}(s)$. One can check that t indeed is a well-defined lambda tree that is $\overline{a}$-guarded and is the least upper bound of D.   ◀

▶ **Proposition 3.5** (glbs of $\trianglelefteq_\bot^{\overline{a}}$). *Every non-empty subset T of $\mathcal{T}_\bot^{\overline{a}}$ has a glb $\bigsqcap T$ in $(\mathcal{T}_\bot^{\overline{a}}, \trianglelefteq_\bot^{\overline{a}})$ such that $\mathcal{D}(\bigsqcap T)$ is the largest set P satisfying the following properties:*

(1) If $p \in P$, then there is some $l \in \mathcal{L}$ such that $s(p) = l$ for all $s \in T$.

(2) If $p \cdot \langle i \rangle \in P$, then $p \in P$.

(3) If $p \in P$, $a_i = 0$, and $p \cdot \langle i \rangle \in \mathcal{D}(s)$ for some $s \in T$, then $p \cdot \langle i \rangle \in P$.

---

[2] Equivalently, complete semilattices are bounded complete cpos. Hence, complete semilattices are a generalisation of *Scott domains* (which in addition have to be *algebraic*).

Proof sketch. Let $P \subseteq \mathcal{P}$ be the largest set satisfying (1) to (3). As these properties are closed under union, P is well-defined. We define the partial function $t \colon \mathcal{P} \rightharpoonup \mathcal{L}$ as the restriction of an arbitrary lambda tree in T to P. Using (1) and (2), one can show that t is indeed a well-defined $\bar{a}$-guarded lambda tree. One can then check that t is the glb of T. ◄

For instance $\bigsqcap\{\lambda\mathsf{x}.\mathsf{x}\,\mathsf{y}, \lambda\mathsf{x}.\mathsf{y}\,\mathsf{x}\}$ is $\lambda\mathsf{x}.\bot\,\bot$ for 011, $\lambda\mathsf{x}.\bot$ for 110, and $\bot$ for 001.

▶ **Theorem 3.6.** $(\mathcal{T}_\bot^{\bar{a}}, \trianglelefteq_\bot^{\bar{a}})$ *is a complete semilattice for any* $\bar{a}$.

Proof. Follows from Theorem 3.4 and Proposition 3.5. ◄

We conclude this section by establishing the partially ordered set $(\mathcal{T}_\bot^{\bar{a}}, \trianglelefteq_\bot^{\bar{a}})$ as (order isomorphic to) the ideal completion of $(\Lambda_\bot, \leq_\bot^{\bar{a}})$. Recall that, given a partially order set $(A, \leq)$, its ideal completion is an extension of the original partially ordered set to a cpo. A set $B \subseteq A$ is called an *ideal* in $(A, \leq)$ if it is directed and *downward-closed*, where the latter means that for all $a \in A, b \in B$ with $a \leq b$, we have that $a \in B$. The *ideal completion* of $(A, \leq)$, is the partially ordered set $(\mathcal{I}, \subseteq)$, where $\mathcal{I}$ is the set of all ideals in $(A, \leq)$ and $\subseteq$ is standard set inclusion.

▶ **Theorem 3.7.** *The ideal completion of* $(\Lambda_\bot, \leq_\bot^{\bar{a}})$ *is order isomorphic to* $(\mathcal{T}_\bot^{\bar{a}}, \trianglelefteq_\bot^{\bar{a}})$.

Proof sketch. By Proposition 3.3, it suffices to show that the ideal completion $(\mathcal{I}, \subseteq)$ of $(\mathcal{T}_\bot, \trianglelefteq_\bot^{\bar{a}})$ is order isomorphic to $(\mathcal{T}_\bot^{\bar{a}}, \trianglelefteq_\bot^{\bar{a}})$. To this end, we define two functions $\phi \colon \mathcal{T}_\bot^{\bar{a}} \to \mathcal{I}$ and $\psi \colon \mathcal{I} \to \mathcal{T}_\bot^{\bar{a}}$ as follows: $\phi(t) = \{s \in \mathcal{T}_\bot \mid s \trianglelefteq_\bot^{\bar{a}} t\}$; $\psi(T) = \bigsqcup T$. Well-definedness of ϕ and ψ follows from König's Lemma and Theorem 3.4, respectively. Both ϕ and ψ are obviously monotone and one can check that ϕ and ψ are inverses of each other. Hence, $(\mathcal{I}, \subseteq)$ is order isomorphic to $(\mathcal{T}_\bot^{\bar{a}}, \trianglelefteq_\bot^{\bar{a}})$ ◄

Now that we have established the connection between $\mathcal{T}_\bot^{\bar{a}}$ and the metric completion resp. the ideal completion of $\Lambda_\bot$, we turn our focus to $\mathcal{T}_\bot^{\bar{a}}$ for the rest of this paper.

The characterisation of lubs and glbs for the complete semilattice $(\mathcal{T}_\bot^{\bar{a}}, \trianglelefteq_\bot^{\bar{a}})$ allows us to relate the corresponding notion of limit inferior with the limit in the complete metric space $(\mathcal{T}_\bot^{\bar{a}}, \mathbf{d}_\mathcal{T}^{\bar{a}})$ as summarised in the following theorem:

▶ **Theorem 3.8.** *Let* $(t_\iota)_{\iota<\alpha}$ *be a sequence in* $\mathcal{T}_\bot^{\bar{a}}$.

(i) *If* $\lim_{\iota\to\alpha} t_\iota = t$ *in* $(\mathcal{T}_\bot^{\bar{a}}, \mathbf{d}_\mathcal{T}^{\bar{a}})$, *then* $\liminf_{\iota\to\alpha} t_\iota = t$ *in* $(\mathcal{T}_\bot^{\bar{a}}, \trianglelefteq_\bot^{\bar{a}})$.
(ii) *If* $\liminf_{\iota\to\alpha} t_\iota = t$ *in* $(\mathcal{T}_\bot^{\bar{a}}, \trianglelefteq_\bot^{\bar{a}})$ *and* t *is total, then* $\lim_{\iota\to\alpha} t_\iota = t$ *in* $(\mathcal{T}_\bot^{\bar{a}}, \mathbf{d}_\mathcal{T}^{\bar{a}})$.

The key to establish the correspondence above is the following characterisation of the limit t of a converging sequence $(t_\iota)_{\iota<\alpha}$ in $(\mathcal{T}_\bot^{\bar{a}}, \mathbf{d}_\mathcal{T}^{\bar{a}})$:

$$\mathcal{D}(t) = \bigcup\nolimits_{\beta<\alpha} \bigcap\nolimits_{\beta\leq\iota<\alpha} \mathcal{D}(t_\iota), \text{ and } \quad t(p) = l \iff \exists\beta<\alpha\forall\beta\leq\iota<\alpha\colon t_\iota(p) = l$$

The proof of the correspondence result makes use of a notion of truncation similar Arnold and Nivat's [1] but generalised to be compatible with the $\trianglelefteq_\bot^{\bar{a}}$-orderings.

From the above findings we can conclude that the limit inferior in $(\mathcal{T}_\bot^{\bar{a}}, \trianglelefteq_\bot^{\bar{a}})$ restricted to total lambda trees coincides with the limit in $(\mathcal{T}^{\bar{a}}, \mathbf{d}_\mathcal{T}^{\bar{a}})$. In other words, the limit inferior is a conservative extension of the limit. In the next section, we transfer this result to (strong) convergence of reductions.

4 Transfinite Reductions

In this section, we study finite and transfinite reductions on lambda trees. To this end, we assume for the remainder of this paper a fixed strictness signature $\bar{a}$ such that all subsequent definitions and theorems work on the same universe of lambda trees $\mathcal{T}_\perp^{\bar{a}}$ and its associated structures $\mathbf{d}_\mathcal{T}^{\bar{a}}$ and $\trianglelefteq_\perp^{\bar{a}}$ (unless stated otherwise). Moreover, we need a suitably general notion of reduction steps beyond the familiar β- and η-rules in order to accommodate Böhm reductions in Section 5.

▶ **Definition 4.1.** A *rewrite system* R is a binary relation on $\mathcal{T}_\perp^{\bar{a}}$ such that $(s,t) \in R$ implies that $s \neq \perp$. Given $s,t \in \mathcal{T}_\perp^{\bar{a}}$ and $p \in \mathcal{P}$, an *R-reduction step* from s to t at p, denoted $s \to_{R,p} t$, is inductively defined as follows: if $(s,t) \in R$, then $s \to_{R,\langle\rangle} t$; if $t \to_{R,p} t'$, then $\lambda x.t \to_{R,\langle 0 \rangle \cdot p} \lambda x.t'$, $t\,s \to_{R,\langle 1 \rangle \cdot p} t'\,s$, and $s\,t \to_{R,\langle 2 \rangle \cdot p} s\,t'$ for all $s \in \mathcal{T}_\perp^{\bar{a}}$. If R or p are irrelevant or clear from the context, we omit them in the notation $\to_{R,p}$. If $(t,t') \in R$, then t is called an *R-redex*. If $s \to_{R,p} t$, then s is said to have an *R-redex occurrence* at p. A lambda tree t is called an *R-normal form* if no *R*-reduction step starts from t. The prefix "*R*-" is dropped if R is irrelevant or clear from the context.

▶ **Example 4.2.** The familiar β- and η-rules form rewrite systems as follows:

$$\beta = \left\{ ((\lambda x.t)\,s, t\,[x/s]) \mid s,t \in \mathcal{T}_\perp^{\bar{a}} \right\} \qquad \eta = \left\{ (\lambda x.t\,x, t) \mid t \in \mathcal{T}_\perp^{\bar{a}}, x \notin \mathsf{Range}(t) \right\}$$

where substitution $t\,[x/s]$ is defined as follows: for each $p \in \mathcal{P}$ we have $t\,[x/s]\,(p) = t(p)$ if $t(p) \in \mathcal{L} \setminus \{x\}$; $t\,[x/s]\,(p) = s(p_2)$ if $p = p_1 \cdot p_2, t(p_1) = x, s(p_2) \in \mathcal{L} \setminus \mathcal{P}$; $t\,[x/s]\,(p) = p_1 \cdot s(p_2)$ if $p = p_1 \cdot p_2, t(p_1) = x, s(p_2) \in \mathcal{P}$; and $t\,[x/s]\,(p)$ is undefined otherwise.

The resulting β-reduction step relation $\to_\beta$ on lambda trees is isomorphic (via the isomorphism of Theorem 3.7) to the lifting of the ordinary finitary β-reduction step relation on lambda terms to the ideal completion via the lifting operator $[\cdot\rangle$ of Blom [8]. An analogous correspondence can be shown for η as well.

▶ **Definition 4.3.** A sequence $S = (t_\iota \to_{R,p_\iota} t_{\iota+1})_{\iota < \alpha}$ of R-reduction steps is called an *R-reduction*; S is called *total* if each t_ι is total. If S is finite, we also write $S \colon t_0 \to_R^* t_\alpha$.

The above notion of reductions is too general as it does not relate lambda trees t_β at a limit ordinal index β to the lambda trees $(t_\iota)_{\iota < \beta}$ that precede it. This shortcoming is addressed with a suitable notion of convergence and continuity. In the literature on infinitary rewriting one finds two different variants of convergence/continuity: a *weak* variant, which defines convergence/continuity only according to the underlying structure (metric limit or limit inferior), and a *strong* variant, which also takes the position of contracted redexes into consideration. While both the metric and the partial order lend themselves to either variant, we only consider the strong variant here and refer the reader to the companion report [5] for the weak variant.

We use the name m-convergence and p-convergence to distinguish between the metric- and the partial order-based notion of convergence, respectively. Our notion of (strong) m-convergence is the same notion of convergence that Kennaway et al. [10] used for their infinitary lambda calculi. For our notion of (strong) p-convergence we instantiate the abstract notion of strong p-convergence from our previous work [2]. The key ingredient of p-convergence is the notion of *reduction context*, which assigns to each reduction step $s \to t$ a lambda tree c with $c \trianglelefteq_\perp^{\bar{a}} s,t$. Intuitively, this reduction context c comprises the (maximal) fragment of s that cannot be changed by the reduction step, regardless of the reduction rule.

For instance, the reduction context of $\lambda x.(\lambda y.y)\,x \to \lambda x.x$ is $\lambda x.\bot$ if $a_0 = 1$, and $\bot$ otherwise. The notion of p-convergence is defined using the limit inferior of the sequence of reduction contexts (instead of the original lambda trees themselves). The canonical approach to derive such a reduction context for any complete semilattice is to take the greatest lower bound of the involved lambda trees s and t that does not contain any position of the redex:

▶ **Definition 4.4.** The *reduction context* of a reduction step $s \to_p t$ is the greatest lambda tree c in $(\mathcal{T}_\bot^{\bar{a}}, \trianglelefteq_\bot^{\bar{a}})$ with $c \trianglelefteq_\bot^{\bar{a}} s, t$ and $p \notin \mathcal{D}(c)$; we write $s \to_c t$ to indicate the reduction context c.

In order to simplify reasoning and provide an intuitive understanding of the concept, we give a direct construction of reduction contexts as well:

▶ **Definition 4.5.** Given $t \in \mathcal{T}_\bot^\infty$ and $p \in \mathcal{D}(t)$, we write $t \setminus p$ for the restriction of t to the domain $\{q \in \mathcal{D}(t) \mid p \not\leq q\}$, and $p\!\downarrow^{\bar{a}}$ for the longest non-strict prefix of p.

That is, $t \setminus p$ is obtained from t by replacing the subtree at p with $\bot$. Moreover, $\downarrow^{\bar{a}}$ can be characterised as follows: $\langle\rangle\!\downarrow^{\bar{a}} = \langle\rangle$; $(p \cdot \langle i \rangle)\!\downarrow^{\bar{a}} = p \cdot \langle i \rangle$ if $a_i = 1$; and $(p \cdot \langle i \rangle)\!\downarrow^{\bar{a}} = p\!\downarrow^{\bar{a}}$ if $a_i = 0$.

▶ **Lemma 4.6.** *The reduction context of $s \to_p t$ is equal to $s \setminus p\!\downarrow^{\bar{a}}$ and $t \setminus p\!\downarrow^{\bar{a}}$.*

Proof sketch. By a straightforward induction on p.   ◀

That is, the reduction context of $s \to_p t$ is obtained from s by removing the most deeply nested subtree that both contains the redex and is in a non-strict position. The ensuing notions of strong convergence of reductions are spelled out as follows:

▶ **Definition 4.7.** An R-reduction $S = (t_\iota \to_{p_\iota, c_\iota} t_{\iota+1})_{\iota < \alpha}$ m-*converges* to t_α, denoted $S \colon t_0 \xrightarrow{m}_R t_\alpha$, if $\lim_{\iota \to \gamma} t_\iota = t_\gamma$ and $(|p_\iota|^{\bar{a}})_{\iota < \gamma}$ tends to infinity for all limit ordinals $\gamma \leq \alpha$. S p-*converges* to t_α, denoted $S \colon t_0 \xrightarrow{p}_R t_\alpha$, if $\liminf_{\iota \to \gamma} c_\iota = t_\gamma$ for all limit ordinals $\gamma \leq \alpha$. S is called m-*continuous* resp. p-*continuous* if the corresponding convergence conditions hold for limit ordinals $\gamma < \alpha$ (instead of $\gamma \leq \alpha$).

Intuitively, strong convergence under-approximates convergence in the underlying structure (i.e. weak convergence) by assuming that every contraction changes the root symbol of the redex. Thus, given a reduction step $s \to_p t$, strong convergence assumes that the shortest position at which s and t differ is p.

The semilattice structure underlying p-convergence ensures that p-continuous reductions always p-converge, whereas m-convergence does not necessarily follow from m-continuity:

▶ **Example 4.8.** Given $\Omega = (\lambda x.x\,x)(\lambda x.x\,x)$ and $t = (\lambda x.x\,\Omega)\,y$, we consider the β-reduction $S \colon t \to t \to \dots$ that repeatedly contracts the redex Ω in t. S is trivially m- and p-continuous. However, it is not m-convergent, since contraction takes place at a constant $\bar{a}$-depth, namely $|\langle 1, 0, 2\rangle|^{\bar{a}}$. But it p-converges to $t \setminus \langle 1, 0, 2\rangle\!\downarrow^{\bar{a}}$, which is also the reduction context of each reduction step in S and is equal to $(\lambda x.x\,\bot)\,y$ if $a_2 = 1$, to $(\lambda x.\bot)\,y$ if $a_2 = 0$ but $a_0 = 1$, to $\bot\,y$ if $\bar{a} = 010$, and to $\bot$ if $\bar{a} = 000$.

Similarly to the correspondence between the limit and the limit inferior in Theorem 3.8, we find a correspondence between p- and m-convergence.

▶ **Proposition 4.9.** *For each reduction $S \colon s \xrightarrow{m} t$, we also have that $S \colon s \xrightarrow{p} t$.*

Proof sketch. Let $S = (t_\iota \to_{p_\iota, c_\iota} t_{\iota+1})_{\iota < \alpha}$. If S m-converges, then $(|p_\iota|^{\overline{a}})_{\iota < \gamma}$ tends to infinity for all limit ordinals $\gamma < \alpha$, i.e. for each $d < \omega$ we have that $|p_\iota|^{\overline{a}} \geq d$ after some $\delta < \gamma$. With the help of Lemma 4.6, one can show that the latter implies that t_ι and c_ι coincide up to $\overline{a}$-depth d for all $\delta \leq \iota < \gamma$. Consequently, $\lim_{\iota \to \gamma} t_\iota = \lim_{\iota \to \gamma} c_\iota$, which, by Theorem 3.8 (i), implies $\lim_{\iota \to \gamma} t_\iota = \liminf_{\iota \to \gamma} c_\iota$. Since this holds for all limit ordinals $\gamma \leq \alpha$, we know that S also p-converges to t.   ◀

With the proposition above, we derive the other direction of the correspondence:

▶ **Proposition 4.10.** $S \colon s \xrightarrow{\text{p}} t$ *implies* $S \colon s \xrightarrow{\text{m}} t$ *whenever* S *and* t *are total.*

Proof sketch. One can show that the totality of S and t implies that the $\overline{a}$-depth of contracted redexes in each open prefix of S tends to infinity. Using Proposition 5.5 from [2], we can show that the latter implies that S also m-converges. Then according to Proposition 4.9, S must m-converge to the same lambda tree t.   ◀

Note that it is not sufficient that the two trees s and t are total. For example, the β-reduction $S \colon (\lambda x.y) \, \Omega \xrightarrow{\text{p}} (\lambda x.y) \perp \to y$ p-converges to y but does not m-converge.

Putting Propositions 4.9 and 4.10 together we obtain that p-convergence is a conservative extension of m-convergence:

▶ **Corollary 4.11.** $S \colon s \xrightarrow{\text{m}} t$ *iff* $S \colon s \xrightarrow{\text{p}} t$ *whenever* S *and* t *are total.*

5   Beta Reduction

So far we have only studied the properties of p-convergence independent of the rewrite system. In this section, we specifically study β-reduction and show infinitary normalisation for all of our calculi, and infinitary confluence for three of them. However, considering pure β-reduction, infinitary confluence only holds for the 111 calculus. We can construct counterexamples for the other calculi:

▶ **Example 5.1** ([10]). Given $a_2 = 0$ and $t = (\lambda x.y) \, \Omega$, we find reductions $t \xrightarrow{\text{p}}_\beta \perp$ and $t \to_\beta y$. Given $a_2 = 1$, $a_1 = 0$, and $t = (\lambda x.x \, y) \, \Omega$, we have $t \xrightarrow{\text{p}}_\beta (\lambda x.x \, y) \perp \to_\beta \perp y$ and $t \to_\beta \Omega \, y \xrightarrow{\text{p}}_\beta \perp$. Similarly, given $a_2 = 1$, $a_0 = 0$, and $t = (\lambda x.\lambda y.x) \, \Omega$, we have $t \xrightarrow{\text{p}}_\beta (\lambda x.\lambda y.x) \perp \to_\beta \lambda y.\perp$ and $t \to_\beta \lambda y.\Omega \xrightarrow{\text{p}}_\beta \perp$.

Infinitary confluence of pure β-reduction fails for all m-convergence calculi of Kennaway et al.[10] – including the 111 calculus. On the other hand, the Böhm reduction calculi of Kennaway et al. [11], which extend pure β-reduction with infinitely many rules of the form $t \to \perp$, do satisfy infinitary confluence for the 001, 101, and 111 calculi.

We would like to obtain similar confluence results for the 001, 101, and 111 p-convergence calculi. However, the gap we have to bridge to achieve infinitary confluence is much narrower in our p-convergence calculi. Intuitively, confluence fails for 001 and 101 because p-convergence only captures partiality that is due to infinite reductions, but not partiality that can propagate via finite reductions: For example, in the 101 calculus we have $\Omega \, y \xrightarrow{\text{p}}_\beta \perp$ but $\perp y \not\xrightarrow{\text{p}}_\beta \perp$. In order to obtain the desired confluence properties, we have to add the rules $\lambda x.\perp \to \perp$ (for 001) and $\perp t \to \perp$ (for 001 and 101). More generally we define these $\mathbb{S}$-rules formally as follows:

$$\mathbb{S} = \left\{ (t_1 \, t_2, \perp) \, \middle| \, t_1, t_2 \in \mathcal{T}_\perp^{\overline{a}}, t_i = \perp, a_i = 0 \right\} \cup \left\{ (\lambda x.\perp, \perp)) \, \middle| \, a_0 = 0 \right\}$$

We use the notation βS to denote $\beta \cup \mathbb{S}$. Abusing notation, we also write $\beta(\mathbb{S})$ to refer to β or βS, e.g. if a property holds for either system. Note that for the 111 calculus, $\beta\mathbb{S} = \beta$.

In addition, we continue studying the relation between m-convergence and p-convergence: In general, they are subtly different, but we show that a p-converging $\beta(\mathbb{S})$-reduction can be adequately simulated by an m-converging $\mathbb{B}$-reduction and vice versa, where $\mathbb{B}$ is an extension of β, called Böhm rewrite system, which additionally contains rules of the form $t \to \bot$. This result uses the same construction used by Kennaway et al. [11] to study so-called *meaningless terms*.

In the remainder of this section we first characterise the set of lambda trees that p-converge to $\bot$ (Section 5.1); we then establish a correspondence between pure p-convergence and m-convergence extended with rules $t \to \bot$ for lambda trees t that p-converge to $\bot$ (Section 5.2); and finally we prove infinitary confluence and normalisation for p-convergent $\beta\mathbb{S}$-reductions in the 001, 101, and 111 calculi (Section 5.3). For the infinitary confluence result, we make use of the correspondence between p-convergence and m-convergence.

5.1   Partiality

We begin with the characterisation of lambda trees that p-converge to $\bot$:

▶ **Definition 5.2.** Given an open reduction $S = (t_\iota \to_{p_\iota} t_{\iota+1})_{\iota<\alpha}$, a position p is called *volatile* in S if, for each $\beta < \alpha$, there is some $\beta \leq \gamma < \alpha$ with $p_\gamma \downarrow^{\bar{a}} \leq p \leq p_\gamma$. If p is volatile in S but no proper prefix of p is, then p is called *outermost-volatile* in S.

For instance, in the β-reduction in Example 4.8, $\langle 1, 0, 2 \rangle$ is volatile and $\langle 1, 0, 2 \rangle \downarrow^{\bar{a}}$ is outermost-volatile. Note that outermost-volatile positions must be non-strict, because if p is volatile, then so is $p \downarrow^{\bar{a}}$.

The presence of volatile positions characterises partiality in p-convergent reductions, which by Corollary 4.11 can be stated as follows:

▶ **Proposition 5.3.** $S \colon s \xrightarrow{\mathsf{m}} t$ iff no prefix of S has volatile positions and $S \colon s \xrightarrow{\mathsf{p}} t$.

Proof sketch. Let $S = (t_\iota \to_{p_\iota} t_{\iota+1})_{\iota<\alpha}$. The "only if" direction follows from Proposition 4.9 and the fact that if $(|p_\iota|^{\bar{a}})_{\iota<\beta}$ tends to infinity, then $S|_\beta$ has no volatile positions. For the "if" direction, the infinite pigeonhole principle yields that $(|p_\iota|^{\bar{a}})_{\iota<\beta}$ tends to infinity. Using this fact, one can show that $S \colon s \xrightarrow{\mathsf{m}} t$.   ◀

More specifically, outermost-volatile positions pinpoint the exact location of partiality in the result of a p-converging reduction.

▶ **Lemma 5.4.** If p is outermost-volatile in $S \colon s \xrightarrow{\mathsf{p}} t$, then $p \in \mathcal{D}_\bot(t)$.

Proof sketch. Let $S = (t_\iota \to_{p_\iota, c_\iota} t_{\iota+1})_{\iota<\alpha}$. Since p is volatile in S, we find for each $\beta < \alpha$ some $\beta \leq \iota < \alpha$ with $p_\iota \downarrow^{\bar{a}} \leq p$. Hence, by Lemma 4.6, we know that $p \notin \mathcal{D}(c_\iota)$. Consequently, by Theorem 3.4 and Proposition 3.5, we have that $p \notin \mathcal{D}(t)$. If $p = \langle \rangle$, then $p \in \mathcal{D}_\bot(t)$ follows immediately. If $p = q \cdot \langle 0 \rangle$, then one can use the fact that no prefix of q is volatile to show that $t(q) = \lambda$, which means that $p \in \mathcal{D}_\bot(t)$. The argument for the cases $p = q \cdot \langle 1 \rangle$ and $p = q \cdot \langle 2 \rangle$ is analogous.   ◀

This characterisation of partiality in terms of volatile positions motivates the following notions of destructiveness and fragility:

▶ **Definition 5.5.** A reduction S is called *destructive* if it is p-continuous and $\langle \rangle$ is volatile in S. A lambda tree $t \in \mathcal{T}_\bot^{\bar{a}}$ is called *fragile* if there is a destructive β-reduction starting from t. The set of all fragile *total* lambda trees is denoted $\mathcal{F}^{\bar{a}}$.

Note that fragility is defined in terms of destructive β-reductions. However, one can show that a destructive β-reduction exists iff a destructive βS-reduction exists.

The following proposition explains why destructive reductions have deserved their name:

▶ **Proposition 5.6.** *An open reduction is destructive iff it p-converges to ⊥.*

Proof sketch. The "only if" direction follows from Lemma 5.4; the converse direction can be shown using the characterisation of the limit inferior (Theorem 3.4, Proposition 3.5).   ◄

For example, the β-reduction $\Omega \to \Omega \to \dots$ (cf. Example 4.8) p-converges to ⊥ and is thus destructive. As a corollary from the above proposition, we obtain that every fragile lambda tree – such as Ω – can be contracted to ⊥ by an open p-convergent reduction.

5.2   Correspondence

To compare m- and p-converging reductions, we employ Böhm rewrite systems and the underlying notion of ⊥-instantiation from Kennaway et al.'s work on meaningless terms [11].

▶ **Definition 5.7.** Let $\mathcal{U} \subseteq \mathcal{T}^\infty$ and $t \in \mathcal{T}^\infty_\perp$. A lambda tree $s \in \mathcal{T}^\infty$ is called a ⊥-instance of t w.r.t. $\mathcal{U}$ if s is obtained from t by inserting elements of $\mathcal{U}$ into t at each position $p \in \mathcal{D}_\perp(t)$, i.e. $s(p) = t(p)$ for all $p \in \mathcal{D}(t)$ and $s|_p \in \mathcal{U}$ for all $p \in \mathcal{D}_\perp(t)$. The set of lambda trees that have a ⊥-instance w.r.t. $\mathcal{U}$ that is in $\mathcal{U}$ itself is denoted $\mathcal{U}_\perp$. In other words, $t \in \mathcal{U}_\perp$ iff there is a lambda tree $s \in \mathcal{U}$ such that s is obtained from t by replacing occurrences of ⊥ in t by lambda trees from $\mathcal{U}$.

In particular, we will use the above construction with the set of fragile total lambda trees $\mathcal{F}^{\overline{a}}$, which gives us the set $\mathcal{F}^{\overline{a}}_\perp$.

Finally, we give the construction of Böhm rewrite systems.

▶ **Definition 5.8.** For each set $\mathcal{U} \subseteq \mathcal{T}^{\overline{a}}$, we define the following two rewrite systems:

$$\bot\!\!\!\bot\,(\mathcal{U}) = \{(t, \perp) \,|\, t \in \mathcal{U}_\perp \setminus \{\perp\}\}\,, \qquad\qquad \mathbb{B}\,(\mathcal{U}) = \beta \cup \bot\!\!\!\bot\,(\mathcal{U})$$

If $\mathcal{U}$ is clear from the context, we instead use the notation $\bot\!\!\!\bot$ and $\mathbb{B}$, respectively.

In particular, we consider the Böhm rewrite system w.r.t. fragile total lambda trees, denoted by $\mathbb{B}\left(\mathcal{F}^{\overline{a}}\right)$. We start with one direction of the correspondence between p-converging β(S)-reductions and m-converging $\mathbb{B}\left(\mathcal{F}^{\overline{a}}\right)$-reductions:

▶ **Theorem 5.9.** *If $s \xrightarrow{\text{p}}\!\!\!\twoheadrightarrow_{\beta\mathbb{S}} t$, then $s \xrightarrow{\text{m}}\!\!\!\twoheadrightarrow_{\mathbb{B}} t$, where $\mathbb{B} = \mathbb{B}\left(\mathcal{F}^{\overline{a}}\right)$.*

Proof sketch. Given $S\colon s \xrightarrow{\text{p}}\!\!\!\twoheadrightarrow_{\beta\mathbb{S}} t$, we construct a $\mathbb{B}$-reduction T from S that also p-converges to t but that has no volatile positions in any of its open prefixes. Thus, according to Proposition 5.3, $T\colon s \xrightarrow{\text{m}}\!\!\!\twoheadrightarrow_{\mathbb{B}} t$. The construction of T removes steps in S that take place at or below any outermost-volatile position of some prefix of S and replaces them by a single $\bot\!\!\!\bot$-step. Such a $\bot\!\!\!\bot$-step can be performed since a fragile lambda tree must be responsible for an outermost-volatile position. Moreover, S-steps in S are $\bot\!\!\!\bot$-steps in T since $\mathbb{S} \subseteq \bot\!\!\!\bot\left(\mathcal{F}^{\overline{a}}\right)$. Lemma 5.4 can then be used to show that the resulting $\mathbb{B}$-reduction T p-converges to t.   ◄

The converse direction of Theorem 5.9 does not hold in general. The problem is that $\bot\!\!\!\bot$-steps can be more selective in which fragile lambda subtree to contract to ⊥ compared to p-convergent reductions with volatile positions. If p is a volatile position, then so is $p{\downarrow}^{\overline{a}}$. Consequently, volatile positions and thus '⊥'s in the result of a p-converging reduction are propagated upwards through strict positions. For example, let $a_0 = 0$, and $t = \lambda\mathsf{y}.\Omega$.

Since Ω is fragile, we have the reduction $t \to_{\perp\!\!\!\perp} \lambda y.\perp$. On the other hand, via p-convergent
β-reductions, t only reduces to itself and $\perp$. This phenomenon, however, does not occur if
we restrict ourselves to the strictness signature 111 or if we only consider $\perp\!\!\!\perp$-normal forms.
Indeed, in the above example, $\lambda y.\perp$ is not a $\perp\!\!\!\perp$-normal form and can be contracted to $\perp$ with
a $\perp\!\!\!\perp$-step.

▶ **Theorem 5.10.** *Let* $\mathbb{B} = \mathbb{B}\left(\mathcal{F}^{\overline{a}}\right)$ *and* $s \xrightarrow{\mathsf{m}}_{\mathbb{B}} t$ *such that* s *is total. Then* $s \xrightarrow{\mathsf{p}}_{\beta} t$ *if* $\overline{a} = 111$
or t *is a* $\perp\!\!\!\perp$-*normal form.*

Proof sketch. The reduction $s \xrightarrow{\mathsf{m}}_{\mathbb{B}} t$ can be factored into $S\colon s \xrightarrow{\mathsf{m}}_{\beta} s'$ and $T\colon s' \xrightarrow{\mathsf{m}}_{\perp\!\!\!\perp} t$ (by the
same proof as Lemma 27 of Kennaway et al. [11]). Moreover, we may assume w.l.o.g. that T
contracts disjoint $\perp\!\!\!\perp$-redexes in s' (using an argument similar to Lemma 7.2.4 of Ketema [12]).
By Proposition 4.9, we have that $S\colon s \xrightarrow{\mathsf{p}}_{\beta} s'$ and that $T\colon s' \xrightarrow{\mathsf{p}}_{\perp\!\!\!\perp} t$. For each step $u \to_{\perp\!\!\!\perp,p} v$ in
T we find a reduction $T_p\colon u \xrightarrow{\mathsf{p}}_{\beta} v'$ in which p is volatile since $u|_p$ must be fragile. Given that
$\overline{a} = 111$ or that t is a $\perp\!\!\!\perp$-normal form, we can show that p is in fact outermost-volatile in T_p.
Hence, the equality $v = v'$ follows from Lemma 5.4. Therefore, we may replace each step
$u \to_{\perp\!\!\!\perp,p} v$ in T by T_p, which yields a reduction $s' \xrightarrow{\mathsf{p}}_{\beta} t$. ◀

That is, in general we get one direction of the correspondence – namely from metric to
partial order reduction – only for reductions to normal forms. However, this does not matter
that much as p-converging $\beta(\mathbb{S})$-reductions (an thus also m-converging $\mathbb{B}\left(\mathcal{F}^{\overline{a}}\right)$-reductions)
are normalising as we show below.

5.3 Infinitary Normalisation and Confluence

We begin by recalling the notion of active lambda trees [11], which we use to establish
infinitary normalisation and as an alternative characterisation of fragile lambda trees (in the
001, 101, and 111 calculi).

▶ **Definition 5.11.** A lambda tree t is called *stable* if no lambda tree t' with $t \to^*_{\beta} t'$ has a
β-redex occurrence at $\overline{a}$-depth 0; t is called *active* if no lambda tree t' with $t \to^*_{\beta} t'$ is stable.
The set of all active *total* lambda trees is denoted by $\mathcal{A}^{\overline{a}}$.

To construct normalising p-convergent reductions, we follow the idea of Kennaway et al. [11]:
We contract all subtrees of the initial lambda tree into stable form. The only way to achieve
this for active subtrees is to annihilate them by a destructive reduction. The basis for that
strategy is the following observation:

▶ **Lemma 5.12.** *Every active lambda tree is fragile.*

Proof. If t_0 is active, we find a reduction $t_0 \to^*_{\beta} t'_0$ to a β-redex at $\overline{a}$-depth 0. By contracting
this redex we get a lambda tree t_1 that is active, too. By repeating this argument we obtain
a destructive reduction $t_0 \to^*_{\beta} t'_0 \to_{\beta} t_1 \to^*_{\beta} t'_1 \to_{\beta} \dots$ ◀

The following normalisation result then follows straightforwardly:

▶ **Theorem 5.13.** *For each* $s \in \mathcal{T}^{\overline{a}}_{\perp}$, *there is a normalising reduction* $s \xrightarrow{\mathsf{p}}_{\beta(\mathbb{S})} t$.

Proof sketch. Similar to Theorem 1 of Kennaway et al. [11]: an active subtree at position
p is by Lemma 5.12 also fragile. Hence, there is a β-reduction in which a prefix of p is
outermost-volatile. By Lemma 5.4, such a reduction annihilates the active subtree at p. This
yields a reduction $s \xrightarrow{\mathsf{p}}_{\beta} t$ to β-normal form t, which can be extended by a reduction $t \xrightarrow{\mathsf{p}}_{\mathbb{S}} u$
to a $\beta\mathbb{S}$-normal form u. ◀

From the above we immediately obtain the corresponding result for m-convergence:

▶ **Theorem 5.14.** *For each $s \in \mathcal{T}_\perp^{\overline{a}}$ there is a normalising reduction $s \xrightarrow{\mathsf{m}}_{\mathbb{B}(\mathcal{F}^{\overline{a}})} t$.*

Proof. By Theorem 5.13 and 5.9, as $\beta\mathbb{S}$-normal forms are also $\mathbb{B}(\mathcal{F}^{\overline{a}})$-normal forms. ◀

Consequently, we can derive the following correspondence result.

▶ **Corollary 5.15.** *For each $s \in \mathcal{T}^{\overline{a}}$ with $s \xrightarrow{\mathsf{m}}_{\mathbb{B}(\mathcal{F}^{\overline{a}})} t$, there is a reduction $t \xrightarrow{\mathsf{m}}_{\mathbb{B}(\mathcal{F}^{\overline{a}})} t'$ such that $s \xrightarrow{\mathsf{p}}_\beta t'$.*

Proof. According to Theorem 5.14, there is a normalising reduction $t \xrightarrow{\mathsf{m}}_{\mathbb{B}(\mathcal{F}^{\overline{a}})} t'$. Then a reduction $s \xrightarrow{\mathsf{p}}_\beta t'$ exists by Theorem 5.10. ◀

A shortcoming of this correspondence property and the correspondence properties established in Section 5.2 is that they consider m-convergence in the system $\mathbb{B}(\mathcal{F}^{\overline{a}})$, which is unsatisfactory since $\mathcal{F}^{\overline{a}}$ is defined using p-convergence. A more appropriate choice would be the set $\mathcal{A}^{\overline{a}}$ of active terms, which is defined in terms of finitary reduction only. To obtain a correspondence in terms of $\mathcal{A}^{\overline{a}}$, we will show that $\mathcal{F}^{\overline{a}} = \mathcal{A}^{\overline{a}}$ for strictness signatures 001, 101, and 111. To prove this equality of fragility and activeness, we need the following key lemma, which can be proved using descendants and complete developments.

▶ **Lemma 5.16** (Infinitary Strip Lemma). *Given $S\colon s \xrightarrow{\mathsf{p}}_{\beta\mathbb{S}} t_1$ and $T\colon s \rightarrow^*_{\beta\mathbb{S}} t_2$, there are reductions $S'\colon t_1 \xrightarrow{\mathsf{p}}_{\beta\mathbb{S}} t$ and $T'\colon t_2 \xrightarrow{\mathsf{p}}_{\beta\mathbb{S}} t$, provided $\overline{a} \in \{001, 101, 111\}$.*

Recall that $\beta\mathbb{S} = \beta$ for $\overline{a} = 111$, i.e. the infinitary strip lemma holds for pure β-reduction in the 111 calculus; but it does not hold for 001 and 101 as Example 5.1 demonstrates. Hence, the need for $\mathbb{S}$-rules. By contrast, in the metric calculi of Kennaway et al. [10] the infinitary strip lemma does not hold for any $\overline{a}$. In order to obtain the infinitary strip lemma and confluence, Kennaway et al. extended β-reduction to Böhm reduction.

We use the Infinitary Strip Lemma to show that p-convergent reductions to $\perp$ can be compressed to length at most ω.

▶ **Lemma 5.17.** *If $\overline{a} \in \{001, 101, 111\}$ and $S\colon t \xrightarrow{\mathsf{p}}_{\beta\mathbb{S}} \perp$, then there is a reduction $T\colon t \xrightarrow{\mathsf{p}}_{\beta\mathbb{S}} \perp$ of length $\leq \omega$. If t is total, then T is a β-reduction of length ω.*

Proof sketch. If $|S| \leq \omega$, we are done. Otherwise, we can construct a finite reduction $t \rightarrow^*_{\beta\mathbb{S}} t'$ with at least one contraction at $\overline{a}$-depth 0 either using a finite approximation property of p-convergence (in case S contracts β-redex at $\overline{a}$-depth 0) or by an induction argument (in case S contracts $\mathbb{S}$-redex at root position). By Lemma 5.16, there is a reduction $S'\colon t' \xrightarrow{\mathsf{p}}_{\beta\mathbb{S}} \perp$. Thus, we can repeat the argument for S'. Iterating this argument yields either a reduction $t \rightarrow^*_{\beta\mathbb{S}} \perp$ or a reduction $t \xrightarrow{\mathsf{p}}_{\beta\mathbb{S}} s'$ of length ω with infinitely many contractions at $\overline{a}$-depth 0, and thus $s' = \perp$. If s is total, then T cannot be finite, as finite $\beta\mathbb{S}$-reductions preserve totality. Hence, no step in T can be an $\mathbb{S}$-step. ◀

▶ **Lemma 5.18.** *If $\overline{a} \in \{001, 101, 111\}$, a total lambda tree is active iff it is fragile.*

Proof. The "only if" direction follows from Lemma 5.12. For the converse direction let t be total and fragile, and let $t \rightarrow^*_\beta t_1$. Since t is fragile, there is a reduction $t \xrightarrow{\mathsf{p}}_{\beta\mathbb{S}} \perp$ according to Proposition 5.6. Hence, by Lemma 5.16, there is a reduction $T\colon t_1 \xrightarrow{\mathsf{p}}_{\beta\mathbb{S}} \perp$, which we can assume, according to Lemma 5.17, to be a β-reduction of length ω. Since T is, by Proposition 5.6, destructive, there is a proper prefix $T'\colon t_1 \xrightarrow{\mathsf{p}}_\beta t_2$ of T such that t_2 has a redex occurrence at $\overline{a}$-depth 0. Because T is of length ω, T' is finite i.e. $T'\colon t_1 \rightarrow^*_\beta t_2$. ◀

The above lemma allows us to derive confluence w.r.t. p-convergent reductions from the confluence results w.r.t. m-convergence of Kennaway et al. [10]:

▶ **Theorem 5.19** (infinitary confluence). *Given $\overline{a} \in \{001, 101, 111\}$, we have that $s \overset{\text{p}}{\twoheadrightarrow}_{\beta \mathbb{S}} t_1$ and $s \overset{\text{p}}{\twoheadrightarrow}_{\beta \mathbb{S}} t_2$ implies that $t_1 \overset{\text{p}}{\twoheadrightarrow}_{\beta \mathbb{S}} t$ and $t_2 \overset{\text{p}}{\twoheadrightarrow}_{\beta \mathbb{S}} t$.*

Proof. According to Theorem 5.13, we can extend the existing reductions by normalising reductions $t_1 \overset{\text{p}}{\twoheadrightarrow}_{\beta \mathbb{S}} t_1'$ and $t_2 \overset{\text{p}}{\twoheadrightarrow}_{\beta \mathbb{S}} t_2'$. By Theorem 5.9 and Lemma 5.18, the resulting normalising reductions $s \overset{\text{p}}{\twoheadrightarrow}_{\beta \mathbb{S}} t_1'$ and $s \overset{\text{p}}{\twoheadrightarrow}_{\beta \mathbb{S}} t_2'$ are also m-convergent $\mathbb{B}\left(\mathcal{A}^{\overline{a}}\right)$-reductions. Kennaway et al. [10] have shown that such reductions are confluent. Hence, $t_1' = t_2'$ (as $\beta \mathbb{S}$-normal forms are $\mathbb{B}\left(\mathcal{A}^{\overline{a}}\right)$-normal forms too). ◀

Together with the earlier normalisation result, this means that the 001, 101, and 111 calculi have unique normal forms w.r.t. $\overset{\text{p}}{\twoheadrightarrow}_{\beta \mathbb{S}}$. By the correspondence results between the metric and the partial order calculi, these normal forms are the same as the unique normal forms w.r.t. $\overset{\text{m}}{\twoheadrightarrow}_{\mathbb{B}\left(\mathcal{A}^{\overline{a}}\right)}$ [10], which in turn correspond to Böhm Trees, Levy-Longo Trees, and Berarducci Trees, respectively.

6 Related Work

The use of ideal completion in lambda calculus to construct infinite terms has a long history (see e.g. Ketema [12] for an overview), in particular in the form of constructing infinite normal forms such as Böhm Trees. In that line of work, the ideal completion is typically based on the fully monotone partial order $\leq_{\perp}$ generated by $\perp \leq_{\perp} M$ for any term M. Different kinds of infinite normal forms are then obtained by modulating the set of rules that are used to generate the normal forms. In this paper, we instead modulated the partial order and we have constructed full infinitary calculi in the style of Kennaway et al. [10]. Blom's abstract theory of infinite normal forms and infinitary rewriting based on ideal completion [8] has been crucial for developing our infinitary calculi.

In previous work, we have compared infinitary rewriting based on partial orders vs. metric spaces in a first-order setting [3, 4]. However, in that work we have only considered fully non-strict convergence, whereas we consider varying modes of strictness in the present paper.

Blom's work [9] on *preservation calculi* is similar to our ideal completion calculi. Blom also considers different calculi indexed by strictness signatures and relates them to the corresponding metric calculi. However, he uses the same partial order $\leq_{\perp}^{111}$ for all calculi; the different calculi vary in the notion of reduction contexts they use. Blom's reduction contexts are the same as our reduction contexts, and his Ω-rules are more general variants of our $\mathbb{S}$-rules. However, his approach of using a single partial order has some caveats:

Firstly, there is no corresponding weak notion of preservation sequences that corresponds to weak m-convergence. Secondly, the partially ordered set $(\mathcal{T}_{\perp}^{\overline{a}}, \leq_{\perp}^{111})$ is only a complete semilattice for $\overline{a} = 111$; otherwise it is not even a cpo and limit inferiors do not always exist. For example, let t be an $\overline{a}$-unguarded lambda tree (i.e. $t \notin \mathcal{T}_{\perp}^{\overline{a}}$), and for each $i < \omega$ let t_i be the restriction of t to positions of depth $< i$, which means that $t_i \in \mathcal{T}_{\perp}^{\overline{a}}$. Then $\liminf_{i \to \omega} t_i$ w.r.t. $\leq_{\perp}^{111}$ is t itself and thus not in $\mathcal{T}_{\perp}^{\overline{a}}$ even though all t_i are. This does not cause a problem, if one only considers reduction contexts of p-continuous reductions, though.

For the comparison of his preservation calculi with the metric calculi, Blom uses a notion of 0-*active* terms, which is different from the notion of active terms as used here and by Kennaway et al. [10, 11] (under the names 0-activeness resp. *abc*-activeness). Blom defines that a lambda tree is 0-active iff there is a destructive reduction of length ω starting from it. 0-activeness is demonstrably different from activeness for any strictness signature with $a_2 = 0$ as Example 5.1 shows. But 0-activeness and activeness do coincide for 001, 101, and 111 as we have shown with the combination of Lemma 5.17 and Lemma 5.18.

─── **References** ───

1   André Arnold and Maurice Nivat. The metric space of infinite trees. algebraic and topological properties. *Fundamenta Informaticae*, 3(4):445–476, 1980.

2   Patrick Bahr. Abstract Models of Transfinite Reductions. In Christopher Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 49–66, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.RTA.2010.49`.

3   Patrick Bahr. Partial order infinitary term rewriting and Böhm trees. In Christopher Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 67–84, Dagstuhl, Germany, 2010. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.RTA.2010.67`.

4   Patrick Bahr. Infinitary term graph rewriting is simple, sound and complete. In Ashish Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA'12)*, volume 15 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 69–84, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.RTA.2012.69`.

5   Patrick Bahr. Ideal completions of the lambda calculus. Companion report, available from the author's web site, 2018.

6   Henk P Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathemantics*. Elsevier Science, revised edition, 1984.

7   Alessandro Berarducci. Infinite λ-calculus and non-sensible models. In A Ursini and P Aglianó, editors, *Logic and algebra*, number 180 in Lecture Notes in Pure and Applied Mathematics, pages 339–378. CRC Press, 1996.

8   Stefan Blom. *Term Graph Rewriting–Syntax and Semantics*. PhD thesis, Vrije Universiteit te Amsterdam, 2001.

9   Stefan Blom. An approximation based approach to infinitary lambda calculi. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications*, volume 3091 of *Lecture Notes in Computer Science*, pages 221–232. Springer Berlin / Heidelberg, 2004. `doi:10.1007/b98160`.

10 Richard Kennaway, Jan Willem Klop, M R Sleep, and Fer-Jan de Vries. Infinitary lambda calculus. *Theoretical Computer Science*, 175(1):93–125, 1997. `doi:DOI:10.1016/S0304-3975(96)00171-5`.

11 Richard Kennaway, Vincent van Oostrom, and Fer-Jan de Vries. Meaningless terms in rewriting. *Journal of Functional and Logic Programming*, 1999(1):1–35, 1999.

12 Jeroen Ketema. *Böhm-Like Trees for Rewriting*. PhD thesis, Vrije Universiteit Amsterdam, 2006. URL: `http://dare.ubvu.vu.nl/handle/1871/9203`.

13 Jean-Jacques Lévy. An algebraic interpretation of the $\lambda\beta K$-calculus; and an application of a labelled λ-calculus. *Theoretical Computer Science*, 2(1):97–114, 1976. `doi:10.1016/0304-3975(76)90009-8`.

14 Jean-Jacques Lévy. *Réductions Correctes et Optimales dans le Lambda-Calcul*. PhD thesis, Université Paris VII, 1978.

15 Giuseppe Longo. Set-theoretical models of λ-calculus: theories, expansions, isomorphisms. *Annals of pure and applied logic*, 24(2):153–188, 1983.

16 Mila E Majster-Cederbaum and Christel Baier. Metric completion versus ideal completion. *Theoretical Computer Science*, 170(1-2):145–171, 1996. `doi:DOI:10.1016/S0304-3975(96)80705-5`.

Term-Graph Anti-Unification

Alexander Baumgartner
Department of Computer Science (DCC), University of Chile, Santiago, Chile
abaumgar@dcc.uchile.cl

Temur Kutsia
Research Institute for Symbolic Computation, Johannes Kepler University Linz, Austria
kutsia@risc.jku.at

Jordi Levy
Artificial Intelligence Research Institute (IIIA), Spanish National Research Council, (CSIC),
Barcelona, Spain
levy@iiia.csic.es

Mateu Villaret
Departament d'Informàtica, Matemàtica Aplicada i Estadística, Universitat de Girona, Girona,
Spain
mateu.villaret@udg.edu

—— Abstract ——

We study anti-unification for possibly cyclic, unranked term-graphs and develop an algorithm,
which computes a minimal complete set of generalizations for them. For bisimilar graphs the
algorithm computes the join in the lattice generated by a functional bisimulation. These results
generalize anti-unification for ranked and unranked terms to the corresponding term-graphs, and
solve also anti-unification problems for rational terms and dags. Our results open a way to widen
anti-unification based code clone detection techniques from a tree representation to a graph
representation of the code.

1 Introduction

Term-graphs are rooted, directed, labeled graphs, which may contain cycles. They can be
used to represent functional expressions compactly and to process them efficiently with
the help of graph transformations. Rewriting with term-graphs has been studied quite
intensively, see, e.g., [4, 6, 15, 19, 20, 27]. Term-graphs can be represented in various ways, for
instance, as constraints [6], hypergraphs [27], systems of recursion equations [4], or arrows in
a category [15]. With cycles, term-graphs can express infinite terms and can model regular
infinite data structures. Some related (not necessarily equivalent) representations, that are
widely used in computer science, include dags, μ-terms, control flow graphs, abstract semantic
graphs, program dependency graphs, certain kinds of flowcharts, process graphs, etc.

In this paper, we study the anti-unification problem for term-graphs: Given two such graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ (maybe with cycles), our goal is to find a graph $\mathcal{G}$, which is a least general common generalization of $\mathcal{G}_1$ and $\mathcal{G}_2$. It means, there should exist variable substitutions σ_1 and σ_2 such that the instances of $\mathcal{G}$ with respect to them, i.e., the graphs $\mathcal{G}\sigma_1$ and $\mathcal{G}\sigma_2$, are equivalent to $\mathcal{G}_1$ and $\mathcal{G}_2$, respectively.

Our representation of term-graphs follows the approach from [4], based on recursion equations. The difference is that we are not restricted to ranked alphabets. Variadic function symbols are permitted and, to take the advantage of such variadicity, hedge variables are used together with individual variables. The latter stands for single graphs, while the former can be instantiated by hedges (finite sequences) of graphs. The equivalence relation is bisimilarity.

It has already been shown in [22] that anti-unification for unranked finite terms is finitary: There are, in general, finitely many least general generalizations (lggs). The same holds for unranked term-graphs, discussed in this paper. We develop an algorithm, which computes such lggs. Equivalence class of a term-graph with respect to bisimilarity is a complete lattice. For bisimilar terms, our algorithm computes the lgg, which is the join in this lattice.

The intuition behind lggs is that they should contain "maximal similarities" between the input graphs and should abstract differences between them by variables uniformly. While this might sound similar to the problem of computing maximal common subgraphs (mcs) between graphs [23, 24], lggs, in general, might contain more edges than mcs's and also give information about differences, which is usually neglected in mcs's.

The results reported in this paper extend our previous results for unranked finite terms [8, 22] to unranked cyclic graphs. In particular, we extend rigid anti-unification from terms to graphs. The rigid version is a more efficient variant of the unranked anti-unification algorithm, since it computes only certain kind of generalizations. It is guided by a rigidity function, which, essentially, decides which nodes of the input graphs should be retained in the generalization. Rigidity function is a parameter of the algorithm. Properties of the latter are proved for arbitrary values of this parameter. As special cases of our results, we obtain anti-unification for ranked term-graphs, rational trees, μ-terms, and dags. To the best of our knowledge, generalization for these structures has not been addressed yet in the literature.

Anti-unification has a pretty wide scope of interesting applications. Originally, it was introduced for inductive reasoning [26]. As a method of computing generalizations, variants of anti-unification are important ingredients of techniques and tools that have found applications in various areas of artificial intelligence, machine learning, reasoning, linguistics, program synthesis, analysis, transformation, verification, etc. We can not give an exhaustive overview of all related work here. A couple of recent references (motivated by different applications) include, e.g., [1, 2, 7, 9, 10, 13, 18, 21, 25]. A particularly interesting motivation comes from software code clone detection, where anti-unification has been successfully incorporated at the level of abstract syntax trees [14, 16, 28]. Our results can serve as a starting point to extend these techniques for graph-based representation of code (e.g., abstract semantic graphs or program dependence graphs) or graph-based languages (e.g., for model transformation). Besides, term-graph anti-unification can be used to construct an index for sets of dags (e.g., substitution tree indexing), which can be useful in declarative programming and reasoning.

The paper is organized as follows: In Sect. 2, we introduce the notions related to unranked term-graphs. Sect. 3 briefly recalls results about term-graph bisimilarity. The notions related to substitutions and generalizations are introduced in Sect. 4. The term-graph generalization algorithm is described in Sect. 5. Conclusions and the future work are discussed in Sect. 6.

An experimental implementation of our anti-unification algorithm can be accessed online: `http://www.risc.jku.at/projects/stout/software/tgau.php`.
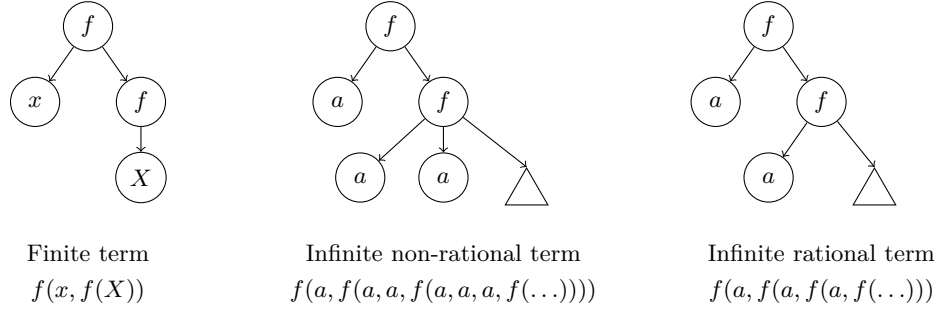
Figure 1 Unranked terms and their tree representations.

2   Unranked Cyclic Term-Graphs

We start by defining unranked (possibly infinite) terms. A *position* $p \in \mathbb{N}^*$ is a sequence of natural numbers. We use a period to separate numbers in a position, e.g. 1.2.3. The empty sequence is denoted by ϵ.

▶ **Definition 1.** Given pairwise disjoint sets of unranked function symbols $\mathcal{F}$ (symbols without fixed arity), term variables $\mathcal{V}_t$, and hedge variables $\mathcal{V}_s$, an *unranked term* is a partial mapping $t : \mathbb{N}^* \to \mathcal{F} \cup \mathcal{V}_t \cup \mathcal{V}_s$ such that

- the domain of t, denoted $dom(t)$, is non-empty and prefix-closed (i.e., if $p_1, p_2 \in \mathbb{N}^*$ and $p_1.p_2 \in dom(t)$, then $p_1 \in dom(t)$),
- for all $p \in \mathbb{N}^*$, if $t(p) \in \mathcal{F}$, then there exists a natural number $n \geq 0$ such that $p.i \in dom(t)$ for all $1 \leq i \leq n$ and $p.i \notin dom(t)$ for all $i > n$,
- for all $p \in \mathbb{N}^*$, if $t(p) \in \mathcal{V}_t \cup \mathcal{V}_s$, then for all n we have $p.n \notin dom(t)$.
- $t(\epsilon) \notin \mathcal{V}_s$.

A term t is *finite* if $dom(t)$ is a finite set. Otherwise it is *infinite*. A term is *rational* if it has finitely many distinct subterms. *Hedges* are finite (possible empty) sequences of terms and hedge variables. The set of terms (respectively, hedges) over $\mathcal{F}$, $\mathcal{V}_t$, and $\mathcal{V}_s$ is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V}_t, \mathcal{V}_s)$ (respectively, $\mathcal{H}(\mathcal{F}, \mathcal{V}_t, \mathcal{V}_s)$). We use the letters f, g, h, a, b, c, and d for function symbols, x, y, z and u for term variables, X, Y, Z, and U for hedge variables, χ, ν, υ and ω for a term variable or a hedge variable, t and r for terms, s and q for a hedge variable or a term, and $\tilde{s}$ and $\tilde{q}$ for hedges. The empty hedge is denoted by ϵ. Given a sequence $\tilde{s}$, the ith element of $\tilde{s}$ is denoted by $\tilde{s}|_i$. Furthermore, $\tilde{s}|_i^j$ denotes the subsequence between the positions i and j where $i < j$, that is, $\tilde{s}|_{i+1}, \dots, \tilde{s}|_{j-1}$. Unranked terms (resp. hedges) can be naturally represented as unranked trees (resp. forests).

▶ **Example 2.** In Fig. 1 we visualize three examples of a finite, infinite non-rational, and infinite rational terms in form of trees. The triangles represent infinite subtrees:

For a term t, we denote by $\mathcal{V}_t(t)$, $\mathcal{V}_s(t)$, and $\mathcal{V}(t)$ respectively the sets of term variables, hedge variables, and all variables occurring in t. The notation extends to hedges as well.

Now we define unranked cyclic term-graphs with the help of recursion equations. We start with a very general notion of a system of recursion equations and subsequently impose restrictions to get to the interesting concept.

▶ **Definition 3.** A *system of recursion equations* over $\mathcal{F}$, $\mathcal{V}_t$, and $\mathcal{V}_s$ is a set of equations $\{x_1 \doteq t_1, \dots, x_n \doteq t_n, X_1 \doteq \tilde{s}_1, \dots, X_m \doteq \tilde{s}_m\}$, where for all i, j, $1 \leq i < j \leq n$, $x_i \neq x_j$, all t's are finite terms, for all i, j, $1 \leq i < j \leq m$, $X_i \neq X_j$, and $\tilde{s}$'s are hedges consisting of finite terms or hedge variables. The variables $x_1, \dots, x_n, X_1, \dots, X_m$ are called *recursion variables*. They are bound in the system. All other variables occurring in the system are free.

We will use different notation for free and bound variables in systems of recursion equations, writing the latter in bold font. One recursion variable (usually, the leading variable of the first equation) is a designated one and we call it the *root* of the system Γ, denoted by $root(\Gamma)$. It is always a term variable.

A recursion variable $\boldsymbol{\nu}$ is *reachable* from a recursion variable $\boldsymbol{\chi}$ in a system Γ if Γ contains an equation of the form $\boldsymbol{\chi} \doteq \tilde{s} \in \Gamma$ and either $\boldsymbol{\nu} \in \mathcal{V}(\tilde{s})$, or $\boldsymbol{\nu}$ is reachable from some recursion variable $\upsilon \in \mathcal{V}(\tilde{s})$. In particular, we say that a hedge variable $\boldsymbol{Y}$ is *horizontally reachable* from a hedge variable $\boldsymbol{X}$ in Γ, if Γ contains an equation $\boldsymbol{X} \doteq \tilde{s}$ such that either $\tilde{s}$ has the form $(\tilde{s}_1, \boldsymbol{Y}, \tilde{s}_2)$, or it has the form $(\tilde{s}_1, \boldsymbol{Z}, \tilde{s}_2)$ and $\boldsymbol{Y}$ is horizontally reachable from $\boldsymbol{Z}$. An equation is called *useless* in Γ if its leading recursion variable is not reachable from $root(\Gamma)$.

A system Γ is called *horizontally bounded* if no hedge variable is reachable from itself in Γ, i.e., Γ contains no horizontal cycle.[1] For instance, $\{\boldsymbol{x} \doteq f(\boldsymbol{x}), \boldsymbol{X} \doteq (\boldsymbol{x}, Y)\}$ is a horizontally bounded system, while $\{\boldsymbol{x} \doteq f(\boldsymbol{x}), \boldsymbol{X} \doteq (\boldsymbol{x}, \boldsymbol{X})\}$ is not.

We do not distinguish between two systems of recursion equations if they differ from each other only by renaming of bound variables.

A system of recursion equations is called *flat*, if the equations have one of the following three possible forms: $\boldsymbol{x} \doteq f(\boldsymbol{\chi}_1, \ldots, \boldsymbol{\chi}_n)$, $\boldsymbol{x} \doteq u$ where u is a free or bound term variable, and $\boldsymbol{X} \doteq (\upsilon_1, \ldots, \upsilon_n)$ where $n \geq 0$ and each υ_i is a free or bound term or hedge variable.

A system of recursion equations Γ is in *canonical form* if it does not contain useless equations and each equation in Γ has one of the following forms:

- $\boldsymbol{x} \doteq f(\boldsymbol{\chi}_1, \ldots, \boldsymbol{\chi}_n)$, where the $\boldsymbol{\chi}$'s are (not necessarily distinct) recursion variables, or
- $\boldsymbol{x} \doteq y$, where y is a free variable, or
- $\boldsymbol{X} \doteq Y$, where Y is a free variable.

For instance, $\{\boldsymbol{x} \doteq f(\boldsymbol{y}, \boldsymbol{X}, \boldsymbol{X}), \boldsymbol{y} \doteq g(\boldsymbol{x}), \boldsymbol{X} \doteq Y\}$ and $\{\boldsymbol{x} \doteq f(\boldsymbol{y}, \boldsymbol{z}, \boldsymbol{z}), \boldsymbol{y} \doteq g(\boldsymbol{x}), \boldsymbol{z} \doteq a\}$ are in canonical form, while $\{\boldsymbol{x} \doteq f(g(\boldsymbol{x}), \boldsymbol{X}), \boldsymbol{X} \doteq Y\}$, $\{\boldsymbol{x} \doteq f(\boldsymbol{y}, X), \boldsymbol{y} \doteq g(\boldsymbol{x})\}$, $\{\boldsymbol{x} \doteq f(\boldsymbol{y}, \boldsymbol{X}), \boldsymbol{y} \doteq g(\boldsymbol{x}), \boldsymbol{X} \doteq a\}$, and $\{\boldsymbol{x} \doteq f(\boldsymbol{y}, \boldsymbol{X}), \boldsymbol{y} \doteq g(\boldsymbol{x}), \boldsymbol{X} \doteq Y, Y \doteq Z\}$ are not.

Every canonical system is flat and horizontally bounded. On the other hand, each flat horizontally bounded system can be transformed to the canonical form by performing the following *canonicalization* steps as long as possible:

- Remove useless equations.
- Remove trivial equations of the form $\boldsymbol{x} \doteq \boldsymbol{y}$ and replace all occurrences of $\boldsymbol{x}$ by $\boldsymbol{y}$. If $\boldsymbol{y} = \boldsymbol{x}$, then replace the equation by $\boldsymbol{x} \doteq \bullet$, where $\bullet$ is some predefined constant from $\mathcal{F}$.
- Replace equations of the form $\boldsymbol{X} \doteq (\upsilon_1, \ldots, \upsilon_n)$, $n > 1$, where υ's are free or bound term or hedge variables, by n new equations $\boldsymbol{Y}_i \doteq \upsilon_i$, $1 \leq i \leq n$, where $\boldsymbol{Y}_i$'s are fresh hedge variables, and replace each occurrence of $\boldsymbol{X}$ by $(\boldsymbol{Y}_1, \ldots, \boldsymbol{Y}_n)$.
- Replace equations of the form $\boldsymbol{X} \doteq u$ by $\boldsymbol{x} \doteq u$ and replace each occurrence of $\boldsymbol{X}$ by $\boldsymbol{x}$, where u is a free or bound term variable and $\boldsymbol{x}$ is a fresh term variable.
- Remove trivial equations of the form $\boldsymbol{X} \doteq Y$ and replace all occurrences of $\boldsymbol{X}$ by $\boldsymbol{Y}$.
- Remove equations of the form $\boldsymbol{X} \doteq \epsilon$ and remove each occurrence of $\boldsymbol{X}$.

Essentially, this canonicalization extends the canonicalization from [4] by four steps dealing with hedge variables. These steps split each equation of the form $\boldsymbol{X} \doteq (s_1, \ldots, s_n)$ into n new equations (one for each s_i), and, eventually only those are retained for which s_i is a free variable. The bound s_i's at the end replace their leading recursion variables.

---

[1] Systems that are not horizontally bounded can be used to define cyclic term-graphs where cycles are formed both vertically and horizontally. Such term-graphs can model infinitely branching trees of infinite depth. These structures go beyond the scope of this paper.

Intuitively, canonical systems of recursion equations can be naturally represented by graphs: The nodes will be the recursion variables; a node $\boldsymbol{x}$ will be connected to a node $\boldsymbol{\chi}$ by an edge if the system contains an equation $\boldsymbol{x} \doteq f(\ldots, \boldsymbol{\chi}, \ldots)$; each node $\boldsymbol{x}$ will have a label f for an equation $\boldsymbol{x} \doteq f(\ldots)$ or the label y for an equation $\boldsymbol{x} \doteq y$, each node $\boldsymbol{X}$ will have a label Y for an equation $\boldsymbol{X} \doteq Y$. Every node is reachable from the root. Cycles and sharings are defined by the occurrences of recursion variables. This intuition justifies the definition:

▶ **Definition 4.** A *term-graph* is a system of recursion equations in canonical form.

▶ **Example 5.** We show some term-graphs and their defining recursion equations.

1. Graph: Equations: $\boldsymbol{x} \doteq f(\boldsymbol{y}, \boldsymbol{x})$
              $\boldsymbol{y} \doteq a$

2. Graph: Equations: $\boldsymbol{x} \doteq f(\boldsymbol{x}, \boldsymbol{y})$
              $\boldsymbol{y} \doteq g(\boldsymbol{y}, \boldsymbol{x})$

3. Graph: Equations: $\boldsymbol{y} \doteq f(\boldsymbol{u}, \boldsymbol{z})$
              $\boldsymbol{z} \doteq f(\boldsymbol{u}, \boldsymbol{y})$
              $\boldsymbol{u} \doteq x$
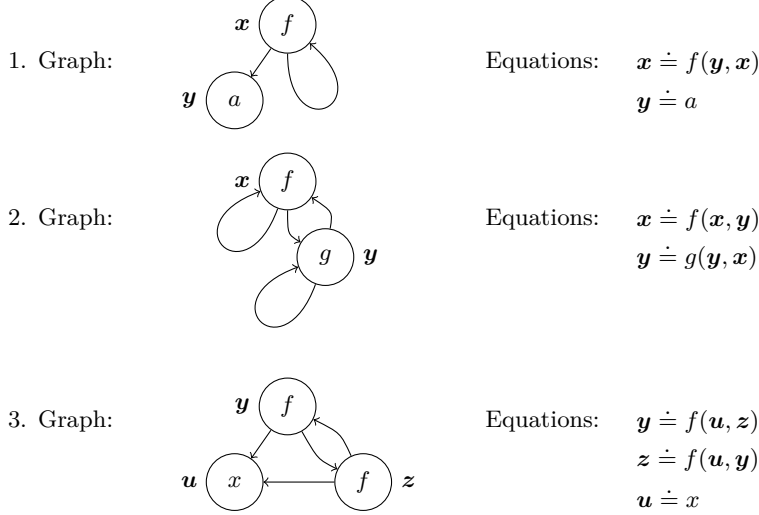
A flat horizontally bounded system and its canonical form have the same (possibly infinite) term unwinding. In the rest of the paper we consider only canonical systems of recursion equations. The words "system of recursion equations" and "(term)-graphs" will be used interchangeably. The letter $\mathcal{G}$ will be used to denote term-graphs.

Given a term-graph $\mathcal{G}$ and an equation $\boldsymbol{x} \doteq t$, the subgraph of $\mathcal{G}$ rooted at $\boldsymbol{x}$ is the set $subgraph(\mathcal{G}, \boldsymbol{x}) = \{\boldsymbol{x} \doteq t\} \cup \{\boldsymbol{\chi} \doteq s \mid \boldsymbol{\chi} \doteq s \in \mathcal{G}, \text{ where } \boldsymbol{\chi} \text{ is reachable from } \boldsymbol{x}\}$. Obviously, $subgraph(\mathcal{G}, root(\mathcal{G})) = \mathcal{G}$.

The set of nodes of a term-graph $\mathcal{G}$ is denoted by $nodes(\mathcal{G})$. If $\boldsymbol{x} \in nodes(\mathcal{G})$ and $\boldsymbol{v}$ is its ith successor (i.e., $\boldsymbol{x} \doteq t \in \mathcal{G}$ for some t and $\boldsymbol{v}$ is ith argument of t), we will write $\boldsymbol{x} \to_i \boldsymbol{v}$. An *access path* of $\boldsymbol{v} \in nodes(\mathcal{G})$ is a possibly empty finite sequence of positive natural numbers $(i_1, \ldots, i_j)$ such that there exist $\boldsymbol{\chi}_1, \ldots, \boldsymbol{\chi}_{j-1} \in nodes(\mathcal{G})$ with $root(\mathcal{G}) \to_{i_1} \boldsymbol{\chi}_1 \to_{i_2} \cdots \to_{i_{j-1}} \boldsymbol{\chi}_{j-1} \to_{i_j} \boldsymbol{v}$. A node may have several access paths. The set of all access paths of a node $\boldsymbol{\chi}$ is denoted by $acc(\boldsymbol{\chi})$.

We will also consider term-graph hedges, defined analogously to hedges: they are finite, possibly empty sequences of term-graphs and hedge variables. We will use $\tilde{\mathcal{G}}$ to denote them.

3 Bisimilarity Relation

It is straightforward to adapt the notions of bisimulation and bisimilarity [4] to our graphs:

▶ **Definition 6.** Let $\Gamma_1 = \{\boldsymbol{\chi}_1 \doteq s_1, \ldots, \boldsymbol{\chi}_n \doteq s_n\}$ and $\Gamma_2 = \{\boldsymbol{v}_1 \doteq q_1, \ldots, \boldsymbol{v}_m \doteq q_m\}$ be two systems of recursion equations. Then R is a *bisimulation* from Γ_1 to Γ_2 iff
- R is a binary relation with the domain $\{\boldsymbol{\chi}_1, \ldots, \boldsymbol{\chi}_n\}$ and codomain $\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m\}$.
- The roots of Γ_1 and Γ_2 are related: $\boldsymbol{\chi}_1 \, R \, \boldsymbol{v}_1$.

- If $\boldsymbol{\chi}_i \, R \, \mathbf{v}_j$, $\boldsymbol{\chi}_i \doteq l_1(\boldsymbol{\chi}_1^i, \ldots, \boldsymbol{\chi}_{k_i}^i) \in \Gamma_1$, $k_i \geq 0$, and $\mathbf{v}_j \doteq l_2(\mathbf{v}_1^j, \ldots, \mathbf{v}_{k_j}^j) \in \Gamma_2$, $k_j \geq 0$, then $l_1 = l_2$, $k_i = k_j$, and $\boldsymbol{\chi}_u \, R \, \mathbf{v}_u$ for all $1 \leq u \leq k_i$. (It applies also when l_1 and l_2 are variables: In this case $k_i = k_j = 0$.)

In short, bisimulation means that the roots are related, related nodes have the same label, and their successor nodes are again related.

▶ **Definition 7.** Two graphs are *bisimilar*, if there exists a bisimulation from one to another.

Bisimilarity is an equivalence relation, see, e.g., [4]. We write $\mathcal{G}_1 \sim \mathcal{G}_2$ if $\mathcal{G}_1$ and $\mathcal{G}_2$ are bisimilar, and $\mathcal{G}_1 \succeq \mathcal{G}_2$ if there exists a functional bisimulation from $\mathcal{G}_1$ to $\mathcal{G}_2$ (i.e., a bisimulation which is a function).

Functional bisimulation collapses a graph into a smaller one. For the other way around, one says that the graph gets expanded, copied, unwinded, or unraveled. In [4] it is shown that the equivalence class of a term-graph $\mathcal{G}$ with respect to bisimilarity is a complete lattice, partially ordered by functional bisimulation. The least upper bound in this lattice is a rational term, denoted by $\triangle \mathcal{G}$, and the greatest lower bound is a fully collapsed graph, denoted by $\triangledown \mathcal{G}$. Hence, $\triangle \mathcal{G} \succeq \mathcal{G} \succeq \triangledown \mathcal{G}$.

▶ **Example 8.** Let $\mathcal{G}$ be the term graph $\{\boldsymbol{x} \doteq f(\boldsymbol{y}, \boldsymbol{z}), \boldsymbol{y} \doteq a, \boldsymbol{z} = f(\boldsymbol{y}, \boldsymbol{x})\}$. Then $\triangle \mathcal{G}$ is the infinite rational term depicted in Fig. 1 in Example 2, and $\triangledown \mathcal{G}$ is the first graph in Example 5.

Given a bisimulation relation R from a term-graph $\mathcal{G}_1$ to a term-graph $\mathcal{G}_2$, its *associated graph* $\mathcal{G}_R^A$ is defined as follows: (i) $nodes(\mathcal{G}_R^A) = R$, $root(\mathcal{G}_R^A) = (root(\mathcal{G}_1), root(\mathcal{G}_2))$, the label of each $(\boldsymbol{\chi}_1, \boldsymbol{\chi}_2) \in nodes(\mathcal{G}_R^A)$ is that of $\boldsymbol{\chi}_1$ (which is the same as the label of $\boldsymbol{\chi}_2$); (ii) if $\boldsymbol{\chi}_1 \in nodes(\mathcal{G}_1)$, $\boldsymbol{\chi}_2 \in nodes(\mathcal{G}_2)$, $(\boldsymbol{\chi}_1, \boldsymbol{\chi}_2) \in R$, $\boldsymbol{\chi}_1 \to_i \boldsymbol{\chi}_1'$, and $\boldsymbol{\chi}_2 \to_i \boldsymbol{\chi}_2'$, then in $\mathcal{G}_R^A$ we have $(\boldsymbol{\chi}_1, \boldsymbol{\chi}_2) \to_i (\boldsymbol{\chi}_1', \boldsymbol{\chi}_2')$.

4 Substitutions and Generalizations

The notions related to substitutions, formulated for finite unranked terms and hedges in [22], can be reused with a slight modification for (possibly) infinite terms and hedges.

A *substitution* is a mapping from term variables to terms and from hedge variables to hedges, which is the identity almost everywhere. We use the traditional finite set representation of substitutions, e.g., $\{x \mapsto f(a, f(a, \ldots)), X \mapsto \epsilon, Y \mapsto (X, g(Y, g(Y, \ldots, Y), Y))\}$, which stands for the substitution that maps every variable to itself except x, X, and Y that are mapped respectively to $f(a, f(a, \ldots))$, ϵ, and $(X, g(Y, g(Y, \ldots, Y), Y))$.

The lower case Greek letters are used to denote substitutions, with the exception of the identity substitution for which we write *Id*. The *domain* and *range* of a substitution σ are defined in the usual way: $dom(\sigma) = \{\chi \in \mathcal{V} \mid \sigma(\chi) \neq \chi\}$ and $ran(\sigma) = \{\sigma(\chi) \mid \chi \in dom(\sigma)\}$.

Substitutions can be *applied* to terms and hedges using the congruences $\sigma(f(s_1, \ldots, s_n)) = f(\sigma(s_1), \ldots, \sigma(s_n))$ and $\sigma(s_1, \ldots, s_n) = (\sigma(s_1), \ldots, \sigma(s_n))$. We call $\sigma(s)$ and $\sigma(\tilde{s})$ the *instances* of respectively s and $\tilde{s}$ and use postfix notation to denote them, writing $s\sigma$ and $\tilde{s}\sigma$. We also say that $\tilde{s}$ is *more general* than $\tilde{q}$ if $\tilde{q}$ is an instance of $\tilde{s}$ and denote this fact by $\tilde{s} \preceq \tilde{q}$. If $\tilde{s} \preceq \tilde{q}$ and $\tilde{q} \preceq \tilde{s}$, then we write $\tilde{s} \simeq \tilde{q}$. If $\tilde{s} \preceq \tilde{q}$ and $\tilde{s} \not\simeq \tilde{q}$, then we say that $\tilde{s}$ is *strictly more general* than $\tilde{q}$ and write $\tilde{s} \prec \tilde{q}$.

The *composition* of two substitutions σ and ϑ, written as $\sigma\vartheta$, is defined as the composition of two mappings: We have $s(\sigma\vartheta) = (s\sigma)\vartheta$ for all s. A substitution σ_1 is *more general* than σ_2 with respect to a set of variables $\mathcal{X} \subseteq \mathcal{V}$, written $\sigma_1 \preceq_{\mathcal{X}} \sigma_2$, if there exists ϑ such that $\chi\sigma_1\vartheta = \chi\sigma_2$, for each $\chi \in \mathcal{X}$. The relations $\simeq$ and $\prec$ are extended to substitutions: $\sigma_1 \simeq_{\mathcal{X}} \sigma_2$ means $\sigma_1 \preceq_{\mathcal{X}} \sigma_2$ and $\sigma_2 \preceq_{\mathcal{X}} \sigma_1$, and $\sigma_1 \prec_{\mathcal{X}} \sigma_2$ means $\sigma_1 \preceq_{\mathcal{X}} \sigma_2$ and $\sigma_1 \not\simeq_{\mathcal{X}} \sigma_2$.

Next we define substitutions directly for term-graphs, i.e., for systems of recursion equations (in canonical form). Instead of writing the whole systems of recursion equations in the range of substitutions, only the roots of the corresponding term-graphs appear there. Hedge variables in the image remain unchanged. For instance, assume the term-graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ are given by the systems of recursion equations: $\mathcal{G}_1 = \{\boldsymbol{x} \doteq f(\boldsymbol{y}, \boldsymbol{x}), \boldsymbol{y} \doteq a\}$, and $\mathcal{G}_2 = \{\boldsymbol{x} \doteq g(\boldsymbol{X}, \boldsymbol{x}, \boldsymbol{X}), \boldsymbol{X} \doteq Y\}$. Then the substitution $\{x \mapsto f(a, f(a, \ldots)), X \mapsto \epsilon, Y \mapsto (X, g(Y, g(Y, \ldots, Y), Y))\}$ we considered above can be written as $\{x \mapsto root(\mathcal{G}_1), X \mapsto \epsilon, Y \mapsto (X, root(\mathcal{G}_2))\}$. The bound variables in $\mathcal{G}_1$ and $\mathcal{G}_2$ should be appropriately renamed to guarantee that the names are distinct from each other and from free variables.

To define application of such a substitution to a term-graph, we assume that all term-graphs are in canonical form and the bound variables are appropriately renamed. Let σ be a substitution and $\mathcal{G}$ be a term-graph. Then the term-graph $\sigma(\mathcal{G})$, the instance of $\mathcal{G}$ under σ, is obtained by canonicalizing the following flat horizontally bounded system of recursion equations: $\{\chi \doteq \sigma(\tilde{s}) \mid \chi \doteq \tilde{s} \in \mathcal{G}\} \cup \mathcal{G}_1 \cup \cdots \cup \mathcal{G}_n$, where $\mathcal{G}_1, \ldots, \mathcal{G}_n$ are all term-graphs whose roots appear in $ran(\sigma)$. Substitution application naturally extends to term-graph hedges.

▶ **Example 9.** Let $\mathcal{G}$ be the term-graph:

$$\mathcal{G} = \{\boldsymbol{x}_0 \doteq f(\boldsymbol{x}_0, \boldsymbol{X}_1, \boldsymbol{x}_1, \boldsymbol{X}_1, \boldsymbol{x}_2), \boldsymbol{X}_1 \doteq X, \boldsymbol{x}_1 \doteq g(\boldsymbol{X}_2, \boldsymbol{x}_2, \boldsymbol{X}_2), \boldsymbol{X}_2 \doteq Y, \boldsymbol{x}_2 \doteq x\}.$$

Let $\sigma = \{x \mapsto root(\mathcal{G}_1), X \mapsto (root(\mathcal{G}_2), X), Y \mapsto \epsilon\}$, $\mathcal{G}_1 = \{\boldsymbol{y} \doteq f(\boldsymbol{y})\}$, $\mathcal{G}_2 = \{\boldsymbol{z} \doteq a\}$. Then

$$\sigma(\mathcal{G}) = \{\boldsymbol{x}_0 \doteq f(\boldsymbol{x}_0, \boldsymbol{z}, \boldsymbol{Z}, \boldsymbol{x}_1, \boldsymbol{z}, \boldsymbol{Z}, \boldsymbol{y}), \boldsymbol{z} \doteq a, \boldsymbol{Z} \doteq X, \boldsymbol{x}_1 \doteq g(\boldsymbol{y}), \boldsymbol{y} \doteq f(\boldsymbol{y})\}.$$

The notion of *more general* term-graphs and term-graph hedges is defined modulo bisimilarity: $\tilde{\mathcal{G}}_1$ is more general than $\tilde{\mathcal{G}}_2$, if there is a substitution σ such that $\tilde{\mathcal{G}}_1\sigma \sim \tilde{\mathcal{G}}_2$. We reuse the symbol $\preceq$ for this relation over term-graphs and term-graph hedges, and also write $\tilde{\mathcal{G}}_1 \simeq \tilde{\mathcal{G}}_2$ if $\tilde{\mathcal{G}}_1 \preceq \tilde{\mathcal{G}}_2$ and $\tilde{\mathcal{G}}_2 \preceq \tilde{\mathcal{G}}_1$. For the strict part of $\preceq$ we reuse $\prec$. Analogously for substitutions: A substitution over term-graphs σ_1 is *more general* than a substitution over term-graphs σ_2 with respect to a set of variables $\mathcal{X} \subseteq \mathcal{V}$, if there exists ϑ such that $\chi\sigma_1\vartheta \sim \chi\sigma_2$ for each $\chi \in \mathcal{X}$. Also in this case we reuse the $\preceq$ symbol and write $\sigma_1 \preceq_{\mathcal{X}} \sigma_2$ (and similarly for the relations $\simeq$ and $\prec$ for substitutions).

A term-graph hedge $\tilde{\mathcal{G}}$ is called a *generalization* of two term-graph hedges $\tilde{\mathcal{G}}_1$ and $\tilde{\mathcal{G}}_2$ if $\tilde{\mathcal{G}} \preceq \tilde{\mathcal{G}}_1$ and $\tilde{\mathcal{G}} \preceq \tilde{\mathcal{G}}_2$. We say that a term-graph $\tilde{\mathcal{G}}$ is a *least general generalization* (lgg in short) of $\tilde{\mathcal{G}}_1$ and $\tilde{\mathcal{G}}_2$ if $\tilde{\mathcal{G}}$ is a generalization of $\tilde{\mathcal{G}}_1$ and $\tilde{\mathcal{G}}_2$ and there is no generalization $\tilde{\mathcal{G}}'$ of $\tilde{\mathcal{G}}_1$ and $\tilde{\mathcal{G}}_2$ that satisfies $\tilde{\mathcal{G}} \prec \tilde{\mathcal{G}}'$. That means, there are no generalizations of $\tilde{\mathcal{G}}_1$ and $\tilde{\mathcal{G}}_2$ that are strictly less general than their least general generalization.

An *anti-unification triple*, AUT in short, is written $\chi : \tilde{\mathcal{G}}_1 \triangleq \tilde{\mathcal{G}}_2$, where χ does not occur in $\tilde{\mathcal{G}}_1$ and $\tilde{\mathcal{G}}_2$. Intuitively, χ is a variable that stands for the most general generalization of $\tilde{\mathcal{G}}_1$ and $\tilde{\mathcal{G}}_2$. An *anti-unifier* of $\chi : \tilde{\mathcal{G}}_1 \triangleq \tilde{\mathcal{G}}_2$ is a substitution σ such that $dom(\sigma) \subseteq \{\chi\}$ and $\chi\sigma$ is a generalization of both $\tilde{\mathcal{G}}_1$ and $\tilde{\mathcal{G}}_2$. An anti-unifier σ of an AUT $\chi : \tilde{\mathcal{G}}_1 \triangleq \tilde{\mathcal{G}}_2$ is *least general (or most specific)* if there is no anti-unifier ϑ of the same problem that satisfies $\sigma \prec_{\{\chi\}} \vartheta$. If σ is a least general anti-unifier of an AUT $\chi : \tilde{\mathcal{G}}_1 \triangleq \tilde{\mathcal{G}}_2$, then $\chi\sigma$ is an lgg of $\tilde{\mathcal{G}}_1$ and $\tilde{\mathcal{G}}_2$.

A *complete set of generalizations* of two term-graph hedges $\tilde{\mathcal{G}}_1$ and $\tilde{\mathcal{G}}_2$ is a set G of term-graph hedges that satisfies the properties:

Soundness: Each $\tilde{\mathcal{G}} \in G$ is a generalization of both $\tilde{\mathcal{G}}_1$ and $\tilde{\mathcal{G}}_2$.
Completeness: For each generalization $\tilde{\mathcal{G}}'$ of $\tilde{\mathcal{G}}_1$ and $\tilde{\mathcal{G}}_2$, there exists $\tilde{\mathcal{G}} \in G$ such that $\tilde{\mathcal{G}}' \preceq \tilde{\mathcal{G}}$.

G is a *minimal complete set of generalizations* (mcsg) of $\tilde{\mathcal{G}}_1$ and $\tilde{\mathcal{G}}_2$ if, in addition to soundness and completeness, it satisfies also the following property:

Minimality: For each $\tilde{\mathcal{G}}'_1, \tilde{\mathcal{G}}'_2 \in G$, if $\tilde{\mathcal{G}}'_1 \preceq \tilde{\mathcal{G}}'_2$ then $\tilde{\mathcal{G}}'_1 = \tilde{\mathcal{G}}'_2$.

▶ **Lemma 10.** *For any hedges $\tilde{s}$ and $\tilde{q}$ there exists their minimal complete set of generalizations. This set is finite and unique modulo $\simeq$.*

Proof. Similar to the analogous lemma for hedges with finite terms, see [22].   ◀

▶ **Theorem 11.** *For any term-graph hedges $\tilde{\mathcal{G}}_1$ and $\tilde{\mathcal{G}}_2$ there exists their minimal complete set of generalizations. This set is finite and unique modulo $\simeq$ and $\sim$.*

Proof. Note that $\mathcal{G} \sim \triangle\mathcal{G}$ for all $\mathcal{G}$. Let $\tilde{\mathcal{G}}_1 = (\mathcal{G}_1^1, \ldots, \mathcal{G}_n^1)$ and $\tilde{\mathcal{G}}_2 = (\mathcal{G}_1^2, \ldots, \mathcal{G}_m^2)$. By Lemma 10, the hedges $(\triangle\mathcal{G}_1^1, \ldots, \triangle\mathcal{G}_n^1)$ and $(\triangle\mathcal{G}_1^2, \ldots, \triangle\mathcal{G}_m^2)$ have a finite minimal complete set of generalizations, unique modulo $\simeq$.   ◀

Our goal is not to compute minimal complete sets of generalizations. We would rather focus on so called rigid generalizations, which we define below. The motivation comes from the experience with finite unranked term anti-unification, where unrestricted mcsg can grow too big and it makes sense to restrict consecutive hedge variables in the generalization. For the details, see [22].[2]

▶ **Definition 12** (Alignment, Rigidity Function). Let w_1 and w_2 be strings of symbols. Then the sequence $a_1[i_1, j_1] \cdots a_n[i_n, j_n]$, for $n \geq 0$, is an *alignment* if i's and j's are positive integers such that $0 < i_1 < \cdots < i_n < |w_1|$ and $0 < j_1 < \cdots < j_n < |w_2|$, and $a_k = w_1|_{i_k} = w_2|_{j_k}$ for all $1 \leq k \leq n$. A *rigidity function* $\mathcal{R}$ is a function that returns, for every pair of strings of symbols w_1 and w_2, a set of alignments of w_1 and w_2.

For instance, if $\mathcal{R}$ computes the set of all longest common subsequences, then $\mathcal{R}(abcda, bcad) = \{b[2,1]c[3,2]a[5,3], \ b[2,1]c[3,2]d[4,4]\}$.

The *top symbol* of a term is defined as $top(x) = x$ for any variable x, and $top(f(\tilde{s})) = f$ for any term $f(\tilde{s})$. The notion is extended to hedges: $top(X) = X$ and $top(s_1, \ldots, s_n) = (top(s_1), \ldots, top(s_n))$. If $\{\chi_1 \doteq s_1, \ldots, \chi_1 \doteq s_n\} \subseteq \mathcal{G}$, $n > 0$, then we define $top(\chi_1, \ldots, \chi_n, \mathcal{G})$ as $top(s_1, \ldots, s_n)$. Moreover, we define $top(\mathcal{G}) = top(root(\mathcal{G}), \mathcal{G})$.

▶ **Definition 13** ($\mathcal{R}$-Generalization). Given two term-graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ (without common free and bound variables) and the rigidity function $\mathcal{R}$, we say that a term-graph $\mathcal{G}$ that generalizes both $\mathcal{G}_1$ and $\mathcal{G}_2$ is their *generalization with respect to $\mathcal{R}$,* or, shortly, an $\mathcal{R}$-generalization, if either

- $\mathcal{R}(top(\mathcal{G}_1), top(\mathcal{G}_2)) \in \{\emptyset, \{\epsilon\}\}$ and $\mathcal{G} = \{x \doteq y\}$, where x is a new bound term variable and y is a new free term variable, or
- $f[1,1] \in \mathcal{R}(top(\mathcal{G}_1), top(\mathcal{G}_2))$ for some f and $\mathcal{G} = \{root(\mathcal{G}) \doteq f(\tilde{\chi})\} \cup \mathcal{Y} \cup \mathcal{G}'_1 \cup \cdots \cup \mathcal{G}'_n$, where $\tilde{\chi}$ does not contain pairs of consecutive hedge recursion variables.
  The sequence $\tilde{\chi}$, the set $\mathcal{Y}$, and the graphs $\mathcal{G}'_1, \ldots, \mathcal{G}'_n$ are defined as follows:
  For $i = 1, 2$, the original graph $\mathcal{G}_i$ contains an equation $root(\mathcal{G}_i) \doteq f(\tilde{v}_i)$ and there exists an alignment $g_1[i_1, j_1] \cdots g_n[i_n, j_n] \in \mathcal{R}(top(\tilde{v}_1, \mathcal{G}_1), top(\tilde{v}_2, \mathcal{G}_2))$, satisfying the following conditions:
  1. If we remove all hedge recursion variables that occur as elements of $\tilde{\chi}$, we get a sequence of term recursion variables $(x_1, \ldots, x_n)$, such that $x_k = root(\mathcal{G}'_k)$ and each $\mathcal{G}'_k$ contains an equation of the form $x_k \doteq g_k(\tilde{v}_k)$ for all $1 \leq k \leq n$, and

---

[2] Note that unrestricted unranked term anti-unification (i.e., without a rigidity function) can be also modeled as associative anti-unification with the unit element. The latter problem has been studied, e.g., in [2, 3].

2. For every $1 \leq k \leq n$, there exists a pair of term recursion variables $\boldsymbol{y}_k^1$ and $\boldsymbol{y}_k^2$ such that $\tilde{\mathbf{v}}_1|_{i_k} = \boldsymbol{y}_k^1$, $\tilde{\mathbf{v}}_2|_{j_k} = \boldsymbol{y}_k^2$, and $\mathcal{G}_k'$ is an $\mathcal{R}$-generalization of $subgraph(\mathcal{G}_1, \boldsymbol{y}_k^1)$ and $subgraph(\mathcal{G}_2, \boldsymbol{y}_k^2)$.

3. $\mathcal{Y} = \{\boldsymbol{Y}_1 \doteq Z_1, \ldots, \boldsymbol{Y}_m \doteq Z_m\}$, where $\boldsymbol{Y}_1, \ldots, \boldsymbol{Y}_m$ are all hedge recursion variables in $\tilde{\chi}$ and $Z_1, \ldots, Z_m$ are new free hedge variables.

▶ **Example 14.** Let $\mathcal{R}$ compute the set of all longest common subsequences and let $\mathcal{G}_1 = \{\boldsymbol{x}_0 \doteq f(\boldsymbol{x}_1, \boldsymbol{x}_2),\ \boldsymbol{x}_1 \doteq g(\boldsymbol{x}_2, \boldsymbol{x}_2),\ \boldsymbol{x}_2 \doteq a\}$ and $\mathcal{G}_2 = \{\boldsymbol{y}_0 \doteq f(\boldsymbol{y}_1, \boldsymbol{y}_0, \boldsymbol{y}_2, \boldsymbol{y}_0),\ \boldsymbol{y}_1 \doteq g,\ \boldsymbol{y}_2 \doteq a\}$. The term graph $\{\boldsymbol{z}_0 \doteq f(\boldsymbol{z}_1, \boldsymbol{Z}_1, \boldsymbol{z}_2, \boldsymbol{Z}_1),\ \boldsymbol{z}_1 \doteq g(\boldsymbol{Z}_2),\ \boldsymbol{z}_2 \doteq a,\ \boldsymbol{Z}_1 \doteq Z_1,\ \boldsymbol{Z}_2 \doteq Z_2\}$ is an $\mathcal{R}$-generalization of $\mathcal{G}_1$ and $\mathcal{G}_2$ while $\{\boldsymbol{z}_0 \doteq f(\boldsymbol{z}_1, \boldsymbol{Z}_1, \boldsymbol{z}_2, \boldsymbol{Z}_1),\ \boldsymbol{z}_1 \doteq g(\boldsymbol{Z}_2, \boldsymbol{Z}_2),\ \boldsymbol{z}_2 \doteq a,\ \boldsymbol{Z}_1 \doteq Z_1,\ \boldsymbol{Z}_2 \doteq Z_2\}$ and $\{\boldsymbol{z}_0 \doteq f(\boldsymbol{z}_1, \boldsymbol{Z}_1, \boldsymbol{z}_2, \boldsymbol{Z}_1),\ \boldsymbol{z}_1 \doteq z,\ \boldsymbol{z}_2 \doteq a,\ \boldsymbol{Z}_1 \doteq Z_1\}$ are not.

5 The Algorithm

We present our anti-unification algorithm as a rule-based algorithm that works on quadruples $A; S; T; \mathcal{G}$, called configurations. Here A, S, and T are sets of anti-unification triples and $\mathcal{G}$ is a term-graph. The rules transform configurations into configurations. Intuitively, the problem set A contains AUTs that have not been solved yet, the store S contains the already solved AUTs, the trail T keeps track of the names of recursion variables, and $\mathcal{G}$ is the generalization which becomes more and more specific as the algorithm progresses by applying the rules.

To keep the notation short, in anti-unification triples we only use variables from the graphs to be generalized. Those graphs are not explicitly present in the configurations, but are global parameters, denoted by $\mathcal{G}_1$ and $\mathcal{G}_2$. For simplicity, we assume that $\mathcal{G}_1$ and $\mathcal{G}_2$ do not contain free variables. This is not a restriction, because we can replace free variables by new constants, use the algorithm defined below, and in the generalization replace those constants back with variables. (In case of hedge variables, we might need to replace their corresponding generalization term variables by generalization hedge variables.) The rigidity function $\mathcal{R}$ is yet another global parameter. In the rules below, generalization term-graphs are assumed to be implicitly transformed into the canonical form.

Step: **Simplification Step**

$\{x : \boldsymbol{y} \triangleq \boldsymbol{z}\} \cup A;\ S;\ T;\ \mathcal{G} \Longrightarrow A_0 \cup A;\ S;\ T \cup \{\boldsymbol{u} : \boldsymbol{y} \triangleq \boldsymbol{z}\};\ \mathcal{G}\{x \mapsto \boldsymbol{u}\} \cup \{\boldsymbol{u} \doteq t\}$,

where $\boldsymbol{y} \doteq l(\tilde{\mathbf{v}}) \in \mathcal{G}_1$, $\boldsymbol{z} \doteq l(\tilde{\mathfrak{v}}) \in \mathcal{G}_2$, $l[1,1] \in \mathcal{R}(top(\boldsymbol{y}, \mathcal{G}_1), top(\boldsymbol{z}, \mathcal{G}_2))$, T does not contain an AUT of the form $\_ : \boldsymbol{y} \triangleq \boldsymbol{z}$, and $\boldsymbol{u}$ is a fresh recursion term variable. If $\tilde{\mathbf{v}} = \tilde{\mathfrak{v}} = \epsilon$ then $t = l$ and $A_0 = \emptyset$, otherwise $t = l(X)$ and $A_0 = \{X : \tilde{\mathbf{v}} \triangleq \tilde{\mathfrak{v}}\}$ where X is a fresh hedge variable.

Dec-S: **Decomposition and Solving**

$\{X : \tilde{\mathbf{v}} \triangleq \tilde{\mathfrak{v}}\} \cup A;\ S;\ T;\ \mathcal{G} \Longrightarrow$

$\quad A \cup \{y_k : \tilde{\mathbf{v}}|_{i_k} \triangleq \tilde{\mathfrak{v}}|_{j_k} \mid 1 \leq k \leq n\};$

$\quad S \cup \{Y_0 : \tilde{\mathbf{v}}|_0^{i_1} \triangleq \tilde{\mathfrak{v}}|_0^{j_1}\} \cup \{Y_k : \tilde{\mathbf{v}}|_{i_k}^{i_{k+1}} \triangleq \tilde{\mathfrak{v}}|_{j_k}^{j_{k+1}} \mid 1 \leq k \leq n-1\} \cup \{Y_n : \tilde{\mathbf{v}}|_{i_n}^{|\tilde{\mathbf{v}}|+1} \triangleq \tilde{\mathfrak{v}}|_{j_n}^{|\tilde{\mathfrak{v}}|+1}\};$

$\quad T;\ \mathcal{G}\sigma \cup \{\boldsymbol{Z}_0 \doteq Y_0, \ldots, \boldsymbol{Z}_n \doteq Y_n\},$

if $\mathcal{R}(top(\tilde{\mathbf{v}}, \mathcal{G}_1), top(\tilde{\mathfrak{v}}, \mathcal{G}_2))$ contains a sequence $l_1[i_1, j_1] \cdots l_n[i_n, j_n]$, $n > 0$. The y's are fresh term variables, the Y's are fresh hedge variables, the $\boldsymbol{Z}$'s are fresh recursion hedge variables, and the substitution is $\sigma = \{X \mapsto (\boldsymbol{Z}_0, y_1, \boldsymbol{Z}_1, \ldots, \boldsymbol{Z}_{n-1}, y_n, \boldsymbol{Z}_n)\}$. For each $1 \leq i \leq n$, if the new AUT has the form $Y_i : \epsilon \triangleq \epsilon$, then it is not added to S and $\boldsymbol{Z}_i$ does not appear in σ.

Solve: **Solving**

$\{\chi : \tilde{\mathbf{v}} \triangleq \tilde{\mathbf{v}}\} \cup A;\ S;\ T;\ \mathcal{G} \Longrightarrow A;\ S \cup \{\chi : \tilde{\mathbf{v}} \triangleq \tilde{\mathbf{v}}\};\ T;\ \mathcal{G}\{\chi \mapsto \boldsymbol{\omega}\} \cup \{\boldsymbol{\omega} \doteq \chi\},$

if $\mathcal{R}(top(\tilde{\mathbf{v}}, \mathcal{G}_1), top(\tilde{\mathbf{v}}, \mathcal{G}_2)) = \emptyset$ or $\mathcal{R}(top(\tilde{\mathbf{v}}, \mathcal{G}_1), top(\tilde{\mathbf{v}}, \mathcal{G}_2)) = \{\epsilon\}$. The variable $\boldsymbol{\omega}$ is a fresh recursion variable. If $\chi \in \mathcal{V}_t$, then $\boldsymbol{\omega} \in \mathcal{V}_t$ and if $\chi \in \mathcal{V}_s$, then $\boldsymbol{\omega} \in \mathcal{V}_s$.

Share: **Sharing**

$\{x : \boldsymbol{y} \triangleq \boldsymbol{z}\} \cup A;\ S;\ \{\boldsymbol{u} : \boldsymbol{y} \triangleq \boldsymbol{z}\} \cup T;\ \mathcal{G} \Longrightarrow A;\ S;\ \{\boldsymbol{u} : \boldsymbol{y} \triangleq \boldsymbol{z}\} \cup T;\ \mathcal{G}\{x \mapsto \boldsymbol{u}\}.$

Merge: **Merging Nodes in the Store**

$\emptyset;\ \{\chi_1 : \tilde{\mathbf{v}} \triangleq \tilde{\mathbf{v}},\ \chi_2 : \tilde{\mathbf{v}} \triangleq \tilde{\mathbf{v}}\} \cup S;\ T;\ \{\boldsymbol{\omega}_1 \doteq \chi_1, \boldsymbol{\omega}_2 \doteq \chi_2\} \cup \mathcal{G} \Longrightarrow$

$\quad \emptyset;\ S \cup \{\chi_1 : \tilde{\mathbf{v}} \triangleq \tilde{\mathbf{v}}\};\ T;\ \mathcal{G}\{\boldsymbol{\omega}_2 \mapsto \boldsymbol{\omega}_1\} \cup \{\boldsymbol{\omega}_1 \doteq \chi_1\},$

where $\chi_1, \chi_2 \in \mathcal{V}_t \cup \mathcal{V}_s$ such that if $\chi_1 \in \mathcal{V}_s$, then $\chi_2 \notin \mathcal{V}_t$.

The rules never generate the AUTs of the form $X : \epsilon \triangleq \epsilon$. To compute $\mathcal{R}$-generalizations of $\mathcal{G}_1$ and $\mathcal{G}_2$, we start with $\{x : root(\mathcal{G}_1) \triangleq root(\mathcal{G}_2)\}, \emptyset, \emptyset, \{\boldsymbol{x} \doteq x\}$ and apply the rules on the selected AUTs in all possible ways. The obtained procedure is denoted by $Gen(\mathcal{R})$.

The notation $\Longrightarrow^*$ abbreviates finite (possible empty) sequence of rule applications. If we want to make it clear which rule is used to transform a configuration, we will write the rule name as the index at the arrow like, e.g., $A; S : T; \mathcal{G} \Longrightarrow_{\mathsf{Step}} A'; S' : T'; \mathcal{G}'$ for the transformation with the rule Simplification Step.
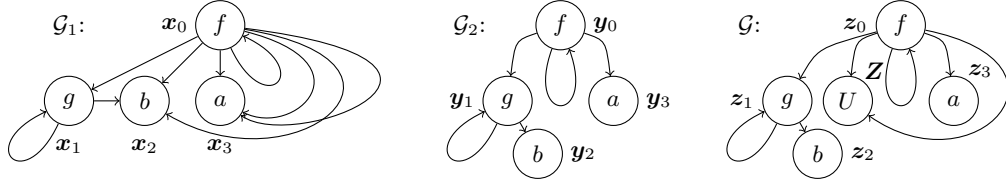
▶ **Example 15.** Let $\mathcal{R}$ be the longest common subsequence. Then the term-graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ below have a unique $\mathcal{R}$-lgg $\mathcal{G}$:

$\mathcal{G}_1 = \{\boldsymbol{x}_0 \doteq f(\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \boldsymbol{x}_0, \boldsymbol{x}_3, \boldsymbol{x}_2, \boldsymbol{x}_3),\ \boldsymbol{x}_1 \doteq g(\boldsymbol{x}_1, \boldsymbol{x}_2),\ \boldsymbol{x}_2 \doteq b,\ \boldsymbol{x}_3 \doteq a\}.$

$\mathcal{G}_2 = \{\boldsymbol{y}_0 \doteq f(\boldsymbol{y}_1, \boldsymbol{y}_0, \boldsymbol{y}_3),\ \boldsymbol{y}_1 \doteq g(\boldsymbol{y}_1, \boldsymbol{y}_2),\ \boldsymbol{y}_2 \doteq b,\ \boldsymbol{y}_3 \doteq a\}.$

$\mathcal{G} = \{\boldsymbol{z}_0 \doteq f(\boldsymbol{z}_1, \boldsymbol{Z}_1, \boldsymbol{z}_0, \boldsymbol{z}_3, \boldsymbol{Z}_1),\ \boldsymbol{z}_1 \doteq g(\boldsymbol{z}_1, \boldsymbol{z}_2),\ \boldsymbol{Z} \doteq U,\ \boldsymbol{z}_3 \doteq a,\ \boldsymbol{z}_2 \doteq b\}.$

Graphically:

The algorithm $Gen(\mathcal{R})$ computes $\mathcal{G}$, e.g., in the following way:

$\{u_0 : \boldsymbol{x}_0 \triangleq \boldsymbol{y}_0\}; \emptyset; \emptyset; \{\boldsymbol{z}_0 \doteq u_0\} \Longrightarrow_{\mathsf{Step}}$

$\{U_0 : (\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \boldsymbol{x}_0, \boldsymbol{x}_3, \boldsymbol{x}_2, \boldsymbol{x}_3) \triangleq (\boldsymbol{y}_1, \boldsymbol{y}_0, \boldsymbol{y}_3)\}; \emptyset; \{\boldsymbol{z}_0 : \boldsymbol{x}_0 \triangleq \boldsymbol{y}_0\};$

$\quad \{\boldsymbol{z}_0 \doteq f(\boldsymbol{Z}_0),\ \boldsymbol{Z}_0 \doteq U_0\} \Longrightarrow_{\mathsf{Dec\text{-}S}}$

(Choosing the common subsequence: $g[1,1]f[4,2]a[5,3]$), corresponding to the

node pairs $\boldsymbol{x}_1$ and $\boldsymbol{y}_1$, $\boldsymbol{x}_0$ and $\boldsymbol{y}_0$, the second occurrence of $\boldsymbol{x}_3$ and $\boldsymbol{y}_3$.)

$\{u_1 : \boldsymbol{x}_1 \triangleq \boldsymbol{y}_1,\ u_2 : \boldsymbol{x}_0 \triangleq \boldsymbol{y}_0,\ u_3 : \boldsymbol{x}_3 \triangleq \boldsymbol{y}_3\}; \{U_1 : (\boldsymbol{x}_2, \boldsymbol{x}_3) \triangleq \epsilon,\ U_2 : (\boldsymbol{x}_2, \boldsymbol{x}_3) \triangleq \epsilon\};$

$\quad \{\boldsymbol{z}_0 : \boldsymbol{x}_0 \triangleq \boldsymbol{y}_0\};$

$\quad \{\boldsymbol{z}_0 \doteq f(\boldsymbol{u}_1, \boldsymbol{Z}_1, \boldsymbol{u}_2, \boldsymbol{u}_3, \boldsymbol{Z}_2),$

$\qquad \boldsymbol{u}_1 \doteq u_1,\ \boldsymbol{Z}_1 \doteq U_1,\ \boldsymbol{u}_2 \doteq u_2,\ \boldsymbol{u}_3 \doteq u_3,\ \boldsymbol{Z}_2 \doteq U_2\} \Longrightarrow_{\mathsf{Step}}$

$\{u_2 : \boldsymbol{x}_0 \triangleq \boldsymbol{y}_0,\ u_3 : \boldsymbol{x}_3 \triangleq \boldsymbol{y}_3,\ U_3 : (\boldsymbol{x}_1, \boldsymbol{x}_2) \triangleq (\boldsymbol{y}_1, \boldsymbol{y}_2)\};$

$\quad \{U_1 : (\boldsymbol{x}_2, \boldsymbol{x}_3) \triangleq \epsilon,\ U_2 : (\boldsymbol{x}_2, \boldsymbol{x}_3) \triangleq \epsilon\};\ \{\boldsymbol{z}_0 : \boldsymbol{x}_0 \triangleq \boldsymbol{y}_0,\ \boldsymbol{z}_1 : \boldsymbol{x}_1 \triangleq \boldsymbol{y}_1\};$

$\{z_0 \doteq f(z_1, Z_1, u_2, u_3, Z_2),$

$\qquad z_1 \doteq g(Z_3), Z_1 \doteq U_1, u_2 \doteq u_2, u_3 \doteq u_3, Z_2 \doteq U_2, Z_3 \doteq U_3\} \Longrightarrow_{\mathsf{Share}}$

$\{u_3 : x_3 \triangleq y_3, U_3 : (x_1, x_2) \triangleq (y_1, y_2)\};$

$\qquad \{U_1 : (x_2, x_3) \triangleq \epsilon, U_2 : (x_2, x_3) \triangleq \epsilon\}; \{z_0 : x_0 \triangleq y_0, z_1 : x_1 \triangleq y_1\};$

$\qquad \{z_0 \doteq f(z_1, Z_1, z_0, u_3, Z_2),$

$\qquad\qquad z_1 \doteq g(Z_3), Z_1 \doteq U_1, u_3 \doteq u_3, Z_2 \doteq U_2, Z_3 \doteq U_3\} \Longrightarrow_{\mathsf{Step}}$

$\{U_3 : (x_1, x_2) \triangleq (y_1, y_2)\};$

$\qquad \{U_1 : (x_2, x_3) \triangleq \epsilon, U_2 : (x_2, x_3) \triangleq \epsilon\}; \{z_0 : x_0 \triangleq y_0, z_1 : x_1 \triangleq y_1, z_3 : x_3 \triangleq y_3\};$

$\qquad \{z_0 \doteq f(z_1, Z_1, z_0, z_3, Z_2),$

$\qquad\qquad z_1 \doteq g(Z_3), Z_1 \doteq U_1, z_3 \doteq a, Z_2 \doteq U_2, Z_3 \doteq U_3\} \Longrightarrow_{\mathsf{Dec\text{-}S}}$

(Choosing the common subsequence: $g[1,1]b[2,2]$), corresponding to the

node pairs x_1 and y_1, x_2 and y_2.)

$\{u_4 : x_1 \triangleq y_1, u_5 : x_2 \triangleq y_2\};$

$\qquad \{U_1 : (x_2, x_3) \triangleq \epsilon, U_2 : (x_2, x_3) \triangleq \epsilon\}; \{z_0 : x_0 \triangleq y_0, z_1 : x_1 \triangleq y_1, z_3 : x_3 \triangleq y_3\};$

$\qquad \{z_0 \doteq f(z_1, Z_1, z_0, z_3, Z_2),$

$\qquad\qquad z_1 \doteq g(u_4, u_5), Z_1 \doteq U_1, z_3 \doteq a, Z_2 \doteq U_2, u_4 \doteq u_4, u_5 \doteq u_5\} \Longrightarrow_{\mathsf{Share}}$

$\{u_5 : x_2 \triangleq y_2\};$

$\qquad \{U_1 : (x_2, x_3) \triangleq \epsilon, U_2 : (x_2, x_3) \triangleq \epsilon\}; \{z_0 : x_0 \triangleq y_0, z_1 : x_1 \triangleq y_1, z_3 : x_3 \triangleq y_3\};$

$\qquad \{z_0 \doteq f(z_1, Z_1, z_0, z_3, Z_2),$

$\qquad\qquad z_1 \doteq g(z_1, u_5), Z_1 \doteq U_1, z_3 \doteq a, Z_2 \doteq U_2, u_5 \doteq u_5\} \Longrightarrow_{\mathsf{Step}}$

$\emptyset; \{U_1 : (x_2, x_3) \triangleq \epsilon, U_2 : (x_2, x_3) \triangleq \epsilon\};$

$\qquad \{z_0 : x_0 \triangleq y_0, z_1 : x_1 \triangleq y_1, z_3 : x_3 \triangleq y_3, z_2 : x_2 \triangleq y_2\};$

$\qquad \{z_0 \doteq f(z_1, Z_1, z_0, z_3, Z_2),$

$\qquad\qquad z_1 \doteq g(z_1, z_2), Z_1 \doteq U_1, z_3 \doteq a, Z_2 \doteq U_2, z_2 \doteq b\} \Longrightarrow_{\mathsf{Merge}}$

$\emptyset; \{U_1 : (x_2, x_3) \triangleq \epsilon\}; \{z_0 : x_0 \triangleq y_0, z_1 : x_1 \triangleq y_1, z_3 : x_3 \triangleq y_3, z_2 : x_2 \triangleq y_2\};$

$\qquad \{z_0 \doteq f(z_1, Z_1, z_0, z_3, Z_1), z_1 \doteq g(z_1, z_2), Z_1 \doteq U_1, z_3 \doteq a, z_2 \doteq b\}.$

The obtained generalization is equal to $\mathcal{G}$ modulo renaming variables. The store and the trail suggest how to obtain the original term-graphs from the computed generalization. For instance, to obtain $\mathcal{G}_1$ from $\mathcal{G}$, we just apply the substitution $\{U_1 \mapsto (x_2, x_3)\}$ to $\mathcal{G}$. In the obtained term-graph we will have $x_2 \doteq b$ and $x_3 \doteq a$ alongside to $z_2 \doteq b$ and $z_3 \doteq a$, but it will be bisimilar to $\mathcal{G}_1$.

▶ **Example 16.** Let $\mathcal{G}_1 = \{x_0 \doteq f(x_1, x_2), x_1 \doteq g(x_0, x_3), x_2 \doteq a, x_3 \doteq b\}$ and $\mathcal{G}_2 = \{y_0 \doteq f(y_1, y_2), y_1 \doteq h(y_0, y_3), y_2 \doteq a, y_3 \doteq b\}$. Then the algorithm ends with $\emptyset, \{z : x_1 \triangleq y_1\}$, $\{z_0 : x_0 \triangleq y_0, z_2 : x_2 \triangleq y_2\}, \{z_0 \doteq f(z_1, z_2), z_1 \doteq z, z_2 \doteq a\}$. Obtaining $\mathcal{G}_1$ from the computed generalization can be illustrated as $\{z_0 \doteq f(z_1, z_2), z_1 \doteq z, z_2 \doteq a\}\{z \mapsto x_1\} = \{z_0 \doteq f(x_1, x_2), z_2 \doteq a, x_1 \doteq g(x_0, x_3), x_0 \doteq f(x_1, x_2), x_2 \doteq a, x_3 \doteq b\} \sim \mathcal{G}_1$.

▶ **Theorem 17** (Termination). *The procedure $Gen(\mathcal{R})$ terminates on any input and produces a configuration $\emptyset; S; T; \mathcal{G}$, where S is irreducible with respect to the merging rule.*

Proof. Let the size of a hedge, $size(\tilde{s})$, be the number of symbols in it. the size of an AUT $x : t_1 \triangleq t_2$ be $size(t_1) + size(t_2) + 1$, and the size of $X : \tilde{s}_1 \triangleq \tilde{s}_2$ be $size(\tilde{s}_1) + size(\tilde{s}_2) + 2$.

The size of a set of AUTs is the multiset of the sizes of its elements. Then the only rule that increases the size of A is Step. However, this step can be applied only finitely many times, since each time it strictly decreases the number of unvisited node pairs (χ_1, χ_2), where $\chi_1 \in \mathcal{G}_1$ and $\chi_2 \in \mathcal{G}_2$. Any other rule strictly decreases the size of A or, in case of Merge, the size of S. Moreover, Merge does not change the size of A. The rule Dec-S can introduce only finite branching. Therefore, the algorithm terminates.   ◀

▶ **Definition 18.** Given a set A of AUTs where all the generalization variables are pairwise distinct. We define two substitutions that can be obtained from A:

$$\sigma_L(A) := \{\chi \mapsto \tilde{\mathbf{v}} \mid \chi : \tilde{\mathbf{v}} \triangleq \tilde{\mathbf{v}} \in A\} \qquad \sigma_R(A) := \{\chi \mapsto \tilde{\mathbf{v}} \mid \chi : \tilde{\mathbf{v}} \triangleq \tilde{\mathbf{v}} \in A\}$$

▶ **Lemma 19** (Transformation Invariant). *Let $\mathcal{G}_1$, $\mathcal{G}_2$ be the two term graphs to be generalized and let $A; S; T; \mathcal{G}$ be a configuration such that all the generalization variables from A, S, T are unique among all the other variables from A, S, T, including those occurring in graphs or hedges. Furthermore, let $\mathcal{G}\sigma_L(T)\sigma_L(S)\sigma_L(A) = \mathcal{G}_1$ and $\mathcal{G}\sigma_R(T)\sigma_R(S)\sigma_R(A) = \mathcal{G}_2$, and let $\mathcal{G}$ be a rigid generalization of $\mathcal{G}$ and $\mathcal{G}_i$ where $i \in \{1, 2\}$.*

If $A; S; T; \mathcal{G} \implies A'; S'; T'; \mathcal{G}'$ is a transformation step applying one of the defined rules then all the generalization variables from A', S', T' are unique among all the other variables from A', S', T'. Moreover, $\mathcal{G}'\sigma_L(T')\sigma_L(S')\sigma_L(A') = \mathcal{G}_1$ and $\mathcal{G}'\sigma_R(T')\sigma_R(S')\sigma_R(A') = \mathcal{G}_2$, and $\mathcal{G}'$ is a rigid generalization of $\mathcal{G}'$ and $\mathcal{G}_i$ where $i \in \{1, 2\}$.

Proof. We prove that each rule preserves those properties. We can omit the proof for $\mathcal{G}'\sigma_R(T')\sigma_R(S')\sigma_R(A') = \mathcal{G}_2$, since it is equivalent to proving $\mathcal{G}'\sigma_L(T')\sigma_L(S')\sigma_L(A') = \mathcal{G}_1$. For the same reason, we omit the proof that $\mathcal{G}'$ is a rigid generalization of $\mathcal{G}'$ and $\mathcal{G}_2$.

In Step we have two cases, namely (i) $\tilde{\mathbf{v}} = \tilde{\mathbf{v}} = \epsilon$, and (ii) $\tilde{\mathbf{v}} \neq \epsilon$ or $\tilde{\mathbf{v}} \neq \epsilon$. We only illustrate the more general case (ii) since the two proofs are largely identical. Therefore, we have $A = \{x : \boldsymbol{y} \triangleq \boldsymbol{z}\} \cup (A' \setminus \{X : \tilde{\mathbf{v}} \triangleq \tilde{\mathbf{v}}\})$, $S = S'$, $T \cup \{\boldsymbol{u} : \boldsymbol{y} \triangleq \boldsymbol{z}\} = T'$, and $\mathcal{G}\{x \mapsto \boldsymbol{u}\} \cup \{\boldsymbol{u} \doteq l(X)\} = \mathcal{G}'$, where $\boldsymbol{y} \doteq l(\tilde{\mathbf{v}}) \in \mathcal{G}_1$, $\boldsymbol{z} \doteq l(\tilde{\mathbf{v}}) \in \mathcal{G}_2$ and $\boldsymbol{u}$, X are fresh. Since $\boldsymbol{u}$, X are fresh, all the generalization variables from A', S', T' are still unique among all the other variables from A', S', T'. Obviously, $\mathcal{G}\sigma_L(T)\sigma_L(S)\sigma_L(A) = \mathcal{G}\sigma_L(T)\sigma_L(S')\sigma_L(\{x : \boldsymbol{y} \triangleq \boldsymbol{z}\} \cup (A' \setminus \{X : \tilde{\mathbf{v}} \triangleq \tilde{\mathbf{v}}\})) = \mathcal{G}_1$. From the uniqueness of x, and by definition of substitution application follows that $\mathcal{G}_1 = \mathcal{G}\{x \mapsto \boldsymbol{y}\}\sigma_L(T)\sigma_L(S')\sigma_L(A' \setminus \{X : \tilde{\mathbf{v}} \triangleq \tilde{\mathbf{v}}\}) = (\mathcal{G}\{x \mapsto \boldsymbol{y}\} \cup \{\boldsymbol{y} \doteq l(\tilde{\mathbf{v}})\})\sigma_L(T)\sigma_L(S')\sigma_L(A' \setminus \{X : \tilde{\mathbf{v}} \triangleq \tilde{\mathbf{v}}\})$. From the uniqueness of X follows $\mathcal{G}_1 = (\mathcal{G}\{x \mapsto \boldsymbol{y}\} \cup \{\boldsymbol{y} \doteq l(X)\})\sigma_L(T)\sigma_L(S')\sigma_L(A')$. Finally, from the uniqueness of $\boldsymbol{u}$ follows $\mathcal{G}_1 = (\mathcal{G}\{x \mapsto \boldsymbol{y}\} \cup \{\boldsymbol{y} \doteq l(X)\})\{\boldsymbol{y} \mapsto \boldsymbol{u}\}\sigma_L(T \cup \{\boldsymbol{u} : \boldsymbol{y} \triangleq \boldsymbol{z}\})\sigma_L(S')\sigma_L(A') = \mathcal{G}'\sigma_L(T')\sigma_L(S')\sigma_L(A')$.

Since Step can't lead to consecutive hedge variables and $l[1, 1] \in \mathcal{R}(top(\boldsymbol{y}, \mathcal{G}_1), top(\boldsymbol{z}, \mathcal{G}_2))$, it follows that $\mathcal{G}'$ is a rigid generalization of $\mathcal{G}'$ and $\mathcal{G}_1$.

Now we analyze Dec-S, which is a bit more involved. We have $A = \{X : \tilde{\mathbf{v}} \triangleq \tilde{\mathbf{v}}\} \cup (A' \setminus \{y_k : \tilde{\mathbf{v}}|_{i_k} \triangleq \tilde{\mathbf{v}}|_{j_k} \mid 1 \leq k \leq n\})$, $S \cup \{Y_0 : \tilde{\mathbf{v}}|_0^{i_1} \triangleq \tilde{\mathbf{v}}|_0^{j_1}\} \cup \{Y_k : \tilde{\mathbf{v}}|_{i_k}^{i_{k+1}} \triangleq \tilde{\mathbf{v}}|_{j_k}^{j_{k+1}} \mid 1 \leq k \leq n-1\} \cup \{Y_n : \tilde{\mathbf{v}}|_{i_n}^{|\tilde{\mathbf{v}}|+1} \triangleq \tilde{\mathbf{v}}|_{j_n}^{|\tilde{\mathbf{v}}|+1}\} = S'$, $T = T'$, and $\mathcal{G}\sigma \cup \{\boldsymbol{Z}_0 \doteq Y_0, \ldots, \boldsymbol{Z}_n \doteq Y_n\} = \mathcal{G}'$, where $\mathcal{R}(top(\tilde{\mathbf{v}}, \mathcal{G}_1), top(\tilde{\mathbf{v}}, \mathcal{G}_2))$ contains a sequence $l_1[i_1, j_1] \cdots l_n[i_n, j_n]$, $n > 0$, the y's, Y's, and $\boldsymbol{Z}$'s are fresh, and $\sigma = \{X \mapsto (\boldsymbol{Z}_0, y_1, \boldsymbol{Z}_1, \ldots, \boldsymbol{Z}_{n-1}, y_n, \boldsymbol{Z}_n)\}$. Since all the variables introduced by the transformation are fresh, all the generalization variables from A', S', T' are still unique. We get $\mathcal{G}\sigma_L(T)\sigma_L(S)\sigma_L(A) = \mathcal{G}\sigma_L(T')\sigma_L(S' \setminus (\{Y_0 : \tilde{\mathbf{v}}|_0^{i_1} \triangleq \tilde{\mathbf{v}}|_0^{j_1}\} \cup \{Y_k : \tilde{\mathbf{v}}|_{i_k}^{i_{k+1}} \triangleq \tilde{\mathbf{v}}|_{j_k}^{j_{k+1}} \mid 1 \leq k \leq n-1\} \cup \{Y_n : \tilde{\mathbf{v}}|_{i_n}^{|\tilde{\mathbf{v}}|+1} \triangleq \tilde{\mathbf{v}}|_{j_n}^{|\tilde{\mathbf{v}}|+1}\}))\sigma_L(\{X : \tilde{\mathbf{v}} \triangleq \tilde{\mathbf{v}}\} \cup (A' \setminus \{y_k : \tilde{\mathbf{v}}|_{i_k} \triangleq \tilde{\mathbf{v}}|_{j_k} \mid 1 \leq k \leq n\})) = \mathcal{G}_1$. By uniqueness of X, follows $\mathcal{G}_1 = \mathcal{G}\{X \mapsto \tilde{\mathbf{v}}\}\sigma_L(T')\sigma_L(S' \setminus (\{Y_0 : \tilde{\mathbf{v}}|_0^{i_1} \triangleq \tilde{\mathbf{v}}|_0^{j_1}\} \cup \{Y_k : \tilde{\mathbf{v}}|_{i_k}^{i_{k+1}} \triangleq \tilde{\mathbf{v}}|_{j_k}^{j_{k+1}} \mid 1 \leq$

$k \leq n-1\} \cup \{Y_n : \tilde{\mathbf{v}}|_{i_n}^{|\tilde{\mathbf{v}}|+1} \triangleq \tilde{\mathbf{v}}|_{j_n}^{|\tilde{\mathbf{v}}|+1}\}))\sigma_L(A' \setminus \{y_k : \tilde{\mathbf{v}}|_{i_k} \triangleq \tilde{\mathbf{v}}|_{j_k} \mid 1 \leq k \leq n\})$. Now observe that $\mathcal{G}\{X \mapsto \tilde{\mathbf{v}}\}$ is equivalent to $\mathcal{G}\{X \mapsto (Y_0, y_1, Y_1, \ldots, Y_{n-1}, y_n, Y_n)\}\{Y_0 \mapsto \tilde{\mathbf{v}}|_0^{i_1}\}\{Y_k \mapsto \tilde{\mathbf{v}}|_{i_k}^{i_{k+1}} \mid 1 \leq k \leq n-1\}\{Y_n \mapsto \tilde{\mathbf{v}}|_{i_n}^{|\tilde{\mathbf{v}}|+1}\}\{y_k \mapsto \tilde{\mathbf{v}}|_{i_k} \mid 1 \leq k \leq n\}$, therefore $\mathcal{G}_1 = \mathcal{G}\{X \mapsto (Y_0, y_1, Y_1, \ldots, Y_{n-1}, y_n, Y_n)\}\{y_k \mapsto \tilde{\mathbf{v}}|_{i_k} \mid 1 \leq k \leq n\}\sigma_L(T')\sigma_L(S')\sigma_L(A' \setminus \{y_k : \tilde{\mathbf{v}}|_{i_k} \triangleq \tilde{\mathbf{v}}|_{j_k} \mid 1 \leq k \leq n\}) = \mathcal{G}\{X \mapsto (Y_0, y_1, Y_1, \ldots, Y_{n-1}, y_n, Y_n)\}\sigma_L(T')\sigma_L(S')\sigma_L(A') = (\mathcal{G}\{X \mapsto (\mathbf{Z}_0, y_1, \mathbf{Z}_1, \ldots, \mathbf{Z}_{n-1}, y_n, \mathbf{Z}_n)\} \cup \{\mathbf{Z}_0 \doteq Y_0, \ldots, \mathbf{Z}_n \doteq Y_n\})\sigma_L(T')\sigma_L(S')\sigma_L(A') = \mathcal{G}'\sigma_L(T')\sigma_L(S')\sigma_L(A')$.

Since Dec-S cannot lead to consecutive hedge variables, it follows that $\mathcal{G}'$ is a rigid generalization of $\mathcal{G}'$ and $\mathcal{G}_1$.

We omit the case of Solve because it is very similar to the case of Step.

In Share we have $A = \{x : \mathbf{y} \triangleq \mathbf{z}\} \cup A'$, $S = S'$, $T = T'$, and $\mathcal{G}\{x \mapsto \mathbf{u}\} = \mathcal{G}'$, where $\{\mathbf{u} : \mathbf{y} \triangleq \mathbf{z}\} \in T$. Uniqueness of generalization variables from A', S', T' is obviously maintained. We get $\mathcal{G}\sigma_L(T)\sigma_L(S)\sigma_L(A) = \mathcal{G}\sigma_L(T')\sigma_L(S')\sigma_L(\{x : \mathbf{y} \triangleq \mathbf{z}\} \cup A') = \mathcal{G}_1$ and by uniqueness of x follows $\mathcal{G}_1 = \mathcal{G}\{x \mapsto \mathbf{y}\}\sigma_L(T')\sigma_L(S')\sigma_L(A')$. The trail $\{\mathbf{u} : \mathbf{y} \triangleq \mathbf{z}\} \in T$ tells us that there is already a recursion variable $\mathbf{u}$ in $\mathcal{G}$ that represents the node $\mathbf{y}$ in $\mathcal{G}_1$. Therefore, instead of substituting x with $\mathbf{y}$ we may as well substitute it with $\mathbf{u}$. This consideration leads to $\mathcal{G}_1 = \mathcal{G}\{x \mapsto \mathbf{u}\}\sigma_L(T')\sigma_L(S')\sigma_L(A')$.

The property that $\mathcal{G}'$ is a rigid generalization of $\mathcal{G}'$ and $\mathcal{G}_1$ is obviously maintained during this transformation.

In Merge we have $A = A' = \emptyset$, $S = \{\chi_2 : \tilde{\mathbf{v}} \triangleq \tilde{\mathbf{v}}\} \cup S'$, $T = T'$, $\mathcal{G} = \{\boldsymbol{\omega}_1 \doteq \chi_1, \boldsymbol{\omega}_2 \doteq \chi_2\} \cup \mathcal{G}''$, and $\mathcal{G}' = \mathcal{G}''\{\boldsymbol{\omega}_2 \mapsto \boldsymbol{\omega}_1\} \cup \{\boldsymbol{\omega}_1 \doteq \chi_1\}$, where $\{\chi_1 : \tilde{\mathbf{v}} \triangleq \tilde{\mathbf{v}}\} \in S'$. We get $\mathcal{G}\sigma_L(T)\sigma_L(S)\sigma_L(\emptyset) = (\{\boldsymbol{\omega}_1 \doteq \chi_1, \boldsymbol{\omega}_2 \doteq \chi_2\} \cup \mathcal{G}'')\sigma_L(T')\sigma_L(\{\chi_2 : \tilde{\mathbf{v}} \triangleq \tilde{\mathbf{v}}\} \cup S') = \mathcal{G}_1$ and by uniqueness of χ_2 follows $\mathcal{G}_1 = (\{\boldsymbol{\omega}_1 \doteq \chi_1, \boldsymbol{\omega}_2 \doteq \chi_2\} \cup \mathcal{G}'')\{\chi_2 \mapsto \tilde{\mathbf{v}}\}\sigma_L(T')\sigma_L(S')$. Since $\sigma_L(S')$ also contains the mapping $\{\chi_1 \mapsto \tilde{\mathbf{v}}\}$ we get $\mathcal{G}_1 = (\{\boldsymbol{\omega}_1 \doteq \chi_1, \boldsymbol{\omega}_2 \doteq \chi_2\} \cup \mathcal{G}'')\{\chi_2 \mapsto \chi_1\}\sigma_L(T')\sigma_L(S') = (\mathcal{G}''\{\boldsymbol{\omega}_2 \mapsto \boldsymbol{\omega}_1\} \cup \{\boldsymbol{\omega}_1 \doteq \chi_1\})\sigma_L(T')\sigma_L(S')$.

The property that $\mathcal{G}'$ is a rigid generalization of $\mathcal{G}'$ and $\mathcal{G}_1$ is maintained because of the condition that from $\chi_1 \in \mathcal{V}_{\mathsf{s}}$ follows $\chi_2 \notin \mathcal{V}_{\mathsf{t}}$ forbids the instantiation of a term variable by a hedge variable. ◄

▶ **Theorem 20** (Soundness). *If $\{x : root(\mathcal{G}_1) \triangleq root(\mathcal{G}_2)\}; \emptyset; \emptyset; \{\mathbf{x} \doteq x\} \Longrightarrow^* \emptyset; S; T; \mathcal{G}$ is a derivation in $Gen(\mathcal{R})$, then $\mathcal{G}$ is an $\mathcal{R}$-generalization of $\mathcal{G}_1$ and $\mathcal{G}_2$.*

Proof. The assumptions of Lemma 19 hold for the initial configuration $\{x : root(\mathcal{G}_1) \triangleq root(\mathcal{G}_2)\}; \emptyset; \emptyset; \{\mathbf{x} \doteq x\}$. Since $Gen(\mathcal{R})$ terminates on any input (Theorem 17), it follows that all the generalization variables from S and T are unique among all the other variables from S and T. Moreover, $\mathcal{G}\sigma_L(T)\sigma_L(S) = \mathcal{G}_1$ and $\mathcal{G}\sigma_R(T)\sigma_R(S) = \mathcal{G}_2$, and $\mathcal{G}$ is a rigid generalization of $\mathcal{G}$ and $\mathcal{G}_i$ where $i \in \{1, 2\}$. Obviously $\mathcal{G}$ is a generalization of $\mathcal{G}_1$ and $\mathcal{G}_2$. To prove that $\mathcal{G}$ is an $\mathcal{R}$-generalization, it remains to show that the recursion from Definition 13 item 2 has been applied exhaustively. This follows from the fact that the store is complete, i.e., $\mathcal{G}\sigma_L(T)\sigma_L(S) = \mathcal{G}_1$ and $\mathcal{G}\sigma_R(T)\sigma_R(S) = \mathcal{G}_2$, and from the condition of the rule Solve that $\mathcal{R}(top(\tilde{\mathbf{v}}, \mathcal{G}_1), top(\tilde{\mathbf{v}}, \mathcal{G}_2))$ is either $\emptyset$ or $\{\epsilon\}$. ◄

▶ **Corollary 21** (Soundness of the Store). *If $\{x : root(\mathcal{G}_1) \triangleq root(\mathcal{G}_2)\}; \emptyset; \emptyset; \{\mathbf{x} \doteq x\} \Longrightarrow^* \emptyset; S; T; \mathcal{G}$ is a derivation in $Gen(\mathcal{R})$, then $\mathcal{G}\sigma_L(T)\sigma_L(S) = \mathcal{G}_1$ and $\mathcal{G}\sigma_R(T)\sigma_R(S) = \mathcal{G}_2$.*

Notice that $Gen(\mathcal{R})$ computes generalizations that do not have free term variables. Therefore, they are not considered in the completeness theorem. However, we show in [12] that this restriction can be lifted by adding an additional transformation rule.

▶ **Theorem 22** (Completeness). *Let $\mathcal{G}$ be an $\mathcal{R}$-generalization of $\mathcal{G}_1$ and $\mathcal{G}_2$. Then $Gen(\mathcal{R})$ computes an $\mathcal{R}$-generalization $\mathcal{G}'$ of $\mathcal{G}_1$ and $\mathcal{G}_2$ such that $\mathcal{G} \preceq \mathcal{G}'$.*

Proof. By our assumption, $\mathcal{G}_1$ and $\mathcal{G}_2$ do not contain free variables. If $\mathcal{G}$ has a form $\{root(\mathcal{G}) \doteq x\}$, then x must be a fresh variable and any generalization computed by $Gen(\mathcal{R})$ satisfies the theorem. Now assume $root(\mathcal{G}) \doteq f(\tilde{\boldsymbol{v}}) \in \mathcal{G}$. Then we should have $root(\mathcal{G}_1) \doteq f(\tilde{\boldsymbol{\chi}}) \in \mathcal{G}_1$ and $root(\mathcal{G}_2) \doteq f(\tilde{\boldsymbol{v}}) \in \mathcal{G}_2$ and we can start the derivation with Step. We can make the next step immediately by Dec-S rule, taking the same alignment (from $\mathcal{R}(top(\tilde{\boldsymbol{\chi}}, \mathcal{G}_1), top(\tilde{\boldsymbol{v}}, \mathcal{G}_2))$) which is used in $f(\tilde{\boldsymbol{v}})$ (since $\mathcal{G}$ is an $\mathcal{R}$-generalization, such an alignment exists). Further, if Merge is applicable, we make this step as long as possible.

After these steps, in the set of new AUTs we will have only those which have counterparts for term variables occurring in $\tilde{\boldsymbol{v}}$. In the store there will be those AUTs which are generalized by hedge variables in $\tilde{\boldsymbol{v}}$. It can be that we merged more variables than it is done in $\tilde{\boldsymbol{v}}$, but it does not harm, since we are going to compute a generalization that is less general than $\mathcal{G}$. The trail will store the seen pair of the nodes (in this case the roots of $\mathcal{G}_1$ and $\mathcal{G}_2$). The generalization graph will contain the equation $root(\mathcal{G}') \doteq f(\tilde{\boldsymbol{\omega}})$, that corresponds to the root equation of $\mathcal{G}$, maybe with more shared variables. The bound term variables from $\tilde{\boldsymbol{v}}$ will have their counterparts in $\tilde{\boldsymbol{\omega}}$, but the equations which correspond to those variables in the current version of $\mathcal{G}'$ will have fresh free variables in the right hand side.

Next, we will pick an AUT in the new configuration. Its generalization variable has a unique counterpart in $\mathcal{G}$, which suggests how to make the next step, basically repeating the reasoning as above, unless the AUT has the form $\boldsymbol{z} : \boldsymbol{x} \triangleq \boldsymbol{y}$ and $\boldsymbol{z}' : \boldsymbol{x} \triangleq \boldsymbol{y}$ is already in the trail. We will use the Sharing rule to make the step. It can be horizontal or vertical sharing.

If it is a horizontal sharing, then it does not matter whether those nodes in $\mathcal{G}$ which correspond to $\boldsymbol{z}$ and $\boldsymbol{z}'$ are shared. If they are, then our construction of $\mathcal{G}'$ at this place directly imitates the structure of $\mathcal{G}$. If they are not, the $\mathcal{G}$ at this place is an expansion of $\mathcal{G}'$, but this operation preserves bisimilarity. In the vertical sharing, in addition to the above considered ones, it is also possible that at this place $\mathcal{G}$ is a collapsed version of $\mathcal{G}'$. But again, bisimilarity is preserved. Note that the construction of our derivation is not influenced by whether a particular node of $\mathcal{G}$ has already been seen or not. They are used to guide the construction, and the same node might guide more than one steps.

Iterating this process, eventually we stop with a generalization $\mathcal{G}'$ such that $\mathcal{G} \preceq \mathcal{G}'$.   ◄

▶ **Theorem 23.** *For two bisimilar term-graphs, $Gen(\mathcal{R})$ computes their join in the lattice generated by functional bisimulation.*

Proof. It is easy to see that our algorithm returns only one answer for bisimilar graphs (since there is no branching at Dec-S rule) and the computed generalization contains no new free variables (the store is empty). Then the set T gives exactly a bisimulation, which justifies bisimilarity between the original term-graphs: $R_T = \{(\boldsymbol{v}, \boldsymbol{v}) \mid \boldsymbol{\chi} : \boldsymbol{v} \triangleq \boldsymbol{v} \in T \text{ for some } \boldsymbol{\chi}\}$. The computed generalization $\mathcal{G}$ is the same as the term-graph $\mathcal{G}_{R_T}^A$ associated to R_T. (The node $\boldsymbol{\chi} \in \mathcal{G}$ can be seen as the node $(\boldsymbol{v}, \boldsymbol{v}) \in \mathcal{G}_{R_T}^A$ for each $\boldsymbol{\chi} : \boldsymbol{v} \triangleq \boldsymbol{v} \in T$.) By construction of T, for each $(\boldsymbol{v}, \boldsymbol{v}) \in R_T$, the access paths are not disjoint: $acc(\boldsymbol{v}) \cap acc(\boldsymbol{v}) \neq \emptyset$ (otherwise there would be a new free variable in the generalization introduced by Dec-S). By Proposition 3.13 in [4], it implies that R_T is a minimal bisimulation. Therefore, from the constructive proof of Theorem 3.19 in [4] we conclude that $\mathcal{G}_{R_T}^A$ (i.e. $\mathcal{G}$) is the join of $\mathcal{G}_1$ and $\mathcal{G}_2$.   ◄

▶ **Example 24.** Let $\mathcal{G}_1 = \{\boldsymbol{x}_0 \doteq f(\boldsymbol{x}_1), \boldsymbol{x}_1 \doteq f(\boldsymbol{x}_2), \boldsymbol{x}_2 \doteq f(\boldsymbol{x}_3), \boldsymbol{x}_3 \doteq f(\boldsymbol{x}_4), \boldsymbol{x}_4 \doteq f(\boldsymbol{x}_5), \boldsymbol{x}_5 \doteq f(\boldsymbol{x}_3)\}$ and $\mathcal{G}_2 = \{\boldsymbol{y}_0 \doteq f(\boldsymbol{y}_1), \boldsymbol{y}_1 \doteq f(\boldsymbol{y}_2), \boldsymbol{y}_2 \doteq f(\boldsymbol{y}_3), \boldsymbol{y}_3 \doteq f(\boldsymbol{y}_4), \boldsymbol{y}_4 \doteq f(\boldsymbol{y}_5), \boldsymbol{y}_5 \doteq f(\boldsymbol{y}_6), \boldsymbol{y}_6 \doteq f(\boldsymbol{y}_7), \boldsymbol{y}_7 \doteq f(\boldsymbol{y}_2)\}$. They are bisimilar. The algorithm computes their lgg $\mathcal{G} = \{\boldsymbol{z}_0 \doteq f(\boldsymbol{z}_1), \boldsymbol{z}_1 \doteq f(\boldsymbol{z}_2), \boldsymbol{z}_2 \doteq f(\boldsymbol{z}_3), \boldsymbol{z}_3 \doteq f(\boldsymbol{z}_4), \boldsymbol{z}_4 \doteq f(\boldsymbol{z}_5), \boldsymbol{z}_5 \doteq f(\boldsymbol{z}_6), \boldsymbol{z}_6 \doteq f(\boldsymbol{z}_7), \boldsymbol{z}_7 \doteq f(\boldsymbol{z}_8), \boldsymbol{z}_8 \doteq f(\boldsymbol{z}_3)\}$. It is the join in the lattice of the bisimilarity class of $\mathcal{G}_1$ and $\mathcal{G}_2$ [4].

6 Conclusion

We have presented an anti-unification algorithm for (unranked) term-graphs, which are given as systems of recursion equations. The algorithm is sound, complete, and terminating, and uses a parameter, called rigidity function. The function selects common edges outgoing from the pair of nodes to be generalized. While longest common subsequence is the most intuitive instance of the rigidity function, the properties of the algorithm hold for any concrete rigid instance of the parameter. As a future work, extending simply typed lambda term anti-unification [11] to cyclic lambda terms [5] would provide a generalization of our results from a first-order language to a higher-order one.

References

1 Hassan Aït-Kaci and Gabriella Pasi. Lattice operations on terms over similar signatures. In *Pre-proceedings of the 27th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'17)*, 2017. URL: `https://arxiv.org/abs/1709.00964`.

2 María Alpuente, Santiago Escobar, Javier Espert, and José Meseguer. ACUOS: A system for modular ACU generalization with subtyping and inheritance. In Fermé and Leite [17], pages 573–581. `doi:10.1007/978-3-319-11558-0`.

3 María Alpuente, Santiago Escobar, Javier Espert, and José Meseguer. A modular order-sorted equational generalization algorithm. *Inf. Comput.*, 235:98–136, 2014. `doi:10.1016/j.ic.2014.01.006`.

4 Zena M. Ariola and Jan Willem Klop. Equational term graph rewriting. *Fundam. Inform.*, 26(3/4):207–240, 1996.

5 Zena M. Ariola and Jan Willem Klop. Lambda calculus with explicit recursion. *Inf. Comput.*, 139(2):154–233, 1997. `doi:10.1006/inco.1997.2651`.

6 Hendrik Pieter Barendregt, Marko C. J. D. van Eekelen, John R. W. Glauert, Richard Kennaway, Marinus J. Plasmeijer, and M. Ronan Sleep. Term graph rewriting. In J. W. de Bakker, A. J. Nijman, and Philip C. Treleaven, editors, *PARLE, Parallel Architectures and Languages Europe, Volume II: Parallel Languages*, volume 259 of *LNCS*, pages 141–158. Springer, 1987. `doi:10.1007/3-540-17945-3`.

7 Adam D. Barwell, Christopher Brown, and Kevin Hammond. Finding parallel functional pearls: Automatic parallel recursion scheme detection in Haskell functions via anti-unification. *Future Generation Comp. Syst.*, 79:669–686, 2018. `doi:10.1016/j.future.2017.07.024`.

8 Alexander Baumgartner. *Anti-Unification Algorithms: Design, Analysis, and Implementation.* PhD thesis, Johannes Kepler University Linz, 2015. Available from `http://www.risc.jku.at/publications/download/risc_5180/phd-thesis.pdf`.

9 Alexander Baumgartner and Temur Kutsia. A library of anti-unification algorithms. In Fermé and Leite [17], pages 543–557. `doi:10.1007/978-3-319-11558-0`.

10 Alexander Baumgartner and Temur Kutsia. Unranked second-order anti-unification. *Inf. Comput.*, 255:262–286, 2017. `doi:10.1016/j.ic.2017.01.005`.

11 Alexander Baumgartner, Temur Kutsia, Jordi Levy, and Mateu Villaret. Higher-order pattern anti-unification in linear time. *J. Autom. Reasoning*, 58(2):293–310, 2017. `doi:10.1007/s10817-016-9383-3`.

12 Alexander Baumgartner, Temur Kutsia, Jordi Levy, and Mateu Villaret. Term-graph anti-unification. RISC Report Series 18-02, Research Institute for Symbolic Computation, Johannes Kepler University Linz, Austria, 2018.

13 Tarek Richard Besold and Enric Plaza. Generalize and blend: Concept blending based on generalization, analogy, and amalgams. In Hannu Toivonen, Simon Colton, Michael

Cook, and Dan Ventura, editors, *Proceedings of the Sixth International Conference on Computational Creativity*, pages 150–157. computationalcreativity.net, 2015. URL: `http://computationalcreativity.net/iccc2015/?page_id=331`.

14   Petr Bulychev and Marius Minea. An evaluation of duplicate code detection using anti-unification. In *Proc. 3rd International Workshop on Software Clones*, 2009.

15   Andrea Corradini and Fabio Gadducci. An algebraic presentation of term graphs, via gs-monoidal categories. *Applied Categorical Structures*, 7(4):299–331, 1999. `doi:10.1023/A:1008647417502`.

16   Rylan Cottrell, Joseph J. C. Chang, Robert J. Walker, and Jörg Denzinger. Determining detailed structural correspondence for generalization tasks. In Ivica Crnkovic and Antonia Bertolino, editors, *6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Int. Symposium on Foundations of Software Engineering*, pages 165–174. ACM, 2007.

17   Eduardo Fermé and João Leite, editors. *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings*, volume 8761 of *Lecture Notes in Computer Science*. Springer, 2014. `doi:10.1007/978-3-319-11558-0`.

18   Adelaine Gelain, Cristiano D. Vasconcellos, Carlos Camarão, and Rodrigo Ribeiro. Type inference for GADTs and anti-unification. In Alberto Pardo and S. Doaitse Swierstra, editors, *Programming Languages - 19th Brazilian Symposium SBLP 2015*, volume 9325 of *LNCS*, pages 16–30. Springer, 2015. `doi:10.1007/978-3-319-24012-1`.

19   Richard Kennaway, Jan Willem Klop, M. Ronan Sleep, and Fer-Jan de Vries. On the adequacy of graph rewriting for simulating term rewriting. *ACM Trans. Program. Lang. Syst.*, 16(3):493–523, 1994. `doi:10.1145/177492.177577`.

20   Jan Willem Klop. Term graph rewriting. In Gilles Dowek, Jan Heering, Karl Meinke, and Bernhard Möller, editors, *Higher-Order Algebra, Logic, and Term Rewriting, Second International Workshop, HOA'95, Selected Papers*, volume 1074 of *LNCS*, pages 1–16. Springer, 1995. `doi:10.1007/3-540-61254-8`.

21   Boris Konev and Temur Kutsia. Anti-unification of concepts in description logic EL. In Chitta Baral, James P. Delgrande, and Frank Wolter, editors, *Principles of Knowledge Representation and Reasoning: Fifteenth International Conference, KR'16*, pages 227–236. AAAI Press, 2016. URL: `http://www.aaai.org/Library/KR/kr16contents.php`.

22   Temur Kutsia, Jordi Levy, and Mateu Villaret. Anti-unification for unranked terms and hedges. *J. Autom. Reasoning*, 52(2):155–190, 2014.

23   Giorgio Levi. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *Calcolo*, 9(4):341, 1973.

24   James J McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software: Practice and Experience*, 12(1):23–34, 1982.

25   Santiago Ontañón and Ali Shokoufandeh. Refinement-based similarity measures for directed labeled graphs. In Ashok K. Goel, M. Belén Díaz-Agudo, and Thomas Roth-Berghofer, editors, *Case-Based Reasoning Research and Development - 24th International Conference, ICCBR'16*, volume 9969 of *LNCS*, pages 311–326. Springer, 2016. `doi:10.1007/978-3-319-47096-2`.

26   Gordon D. Plotkin. A note on inductive generalization. *Machine Intell.*, 5(1):153–163, 1970.

27   Detlef Plump. Term graph rewriting. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter 1, pages 3–61. World Scientific, 1999.

28   Simon J. Thompson, Huiqing Li, and Andreas Schumacher.   The pragmatics of clone detection and elimination. *Programming Journal*, 1(2):8, 2017. `doi:10.22152/programming-journal.org/2017/1/8`.

Proof Nets for Bi-Intuitionistic Linear Logic

Gianluigi Bellin

Università di Verona, Verona, Italy
gianluigi.bellin@univr.it

Willem B. Heijltjes

University of Bath, Bath, United Kingdom
w.b.heijltjes@bath.ac.uk

── Abstract ──

Bi-Intuitionistic Linear Logic (BILL) is an extension of Intuitionistic Linear Logic with a par, dual to the tensor, and subtraction, dual to linear implication. It is the logic of categories with a monoidal closed and a monoidal co-closed structure that are related by linear distributivity, a strength of the tensor over the par. It conservatively extends Full Intuitionistic Linear Logic (FILL), which includes only the par.

We give proof nets for the multiplicative, unit-free fragment MBILL-. Correctness is by local rewriting in the style of Danos contractibility, which yields sequentialization into a relational sequent calculus extending the existing one for FILL. We give a second, geometric correctness condition combining Danos-Regnier switching and Lamarche's Essential Net criterion, and demonstrate composition both inductively and as a one-off global operation.

1 Introduction

Obtaining good proof-theoretic characterizations of FILL [17], intuitionistic linear logic with a "par" connective dual to the tensor, and BILL, which further adds "subtract" dual to linear implication, has proved difficult. The main challenge is in combining par, whose natural home is a multi-conclusion calculus, and linear implication, which is most naturally expressed by a single-conclusion calculus. The dual situation holds for tensor and subtraction (below on the right), where tensor naturally prefers multiple assumptions, but subtraction a single assumption. These are the natural sequent rules:

$$\frac{\Gamma \vdash \Delta \; C \; D}{\Gamma \vdash \Delta \; C \wp D} \qquad \frac{\Gamma \; A \vdash B}{\Gamma \vdash A \multimap B} \qquad \frac{A \; B \; \Gamma \vdash \Delta}{A \otimes B \; \Gamma \vdash \Delta} \qquad \frac{D \vdash C \; \Delta}{D - C \vdash \Delta}$$

A system with the above rules, however, does not satisfy cut-elimination [22, 3]: the single-conclusion and single-assumption rules for linear implication and subtraction are too restrictive. But their multi-conclusion and multi-assumption variants,

$$\frac{\Gamma \; A \vdash B \; \Delta}{\Gamma \vdash A \multimap B \; \Delta} \qquad \frac{\Gamma \; D \vdash C \; \Delta}{\Gamma \; D - C \vdash \Delta}$$

are unsound: they collapse the logic into MLL, since mapping linear implication $A \multimap B$ onto $A^{\perp} \wp B$ and subtraction $D - C$ onto $D \otimes C^{\perp}$ preserves provability (in both directions) [6]. Intermediate ground between these variants is found by annotating the rules with a *relation* between the antecedent and the consequent, and requiring that the discharged assumption A in a rule introducing $A \multimap B$ is not related to any additional conclusions Δ (and dually for $D - C$). With this side-condition, and without describing the development of the relation R into S, the rules are as below. The sequent calculus (for FILL) with relational annotation enjoys cut-elimination [4, 11].

$$\frac{\Gamma \ A \vdash_R B \ \Delta}{\Gamma \vdash_S A \multimap B \ \Delta} \ (A \!\!\not R \Delta) \qquad\qquad \frac{\Gamma \ D \vdash_R C \ \Delta}{\Gamma \ D - C \vdash_S \Delta} \ (\Gamma \!\!\not R C)$$

Traditionally, the sequent calculus is a meta-calculus, describing the construction of natural deduction proofs. For linear logic, naturally described in sequent style, the question of what underlying proof objects were constructed led to the development of proof nets [12]. In this paper we ask the same question for BILL: what are the underlying, canonical proof objects of BILL?

Our answer is a notion of proof nets, presented as a graph-like natural deduction calculus, that embodies the perfect duality between tensor and par, and between implication and subtraction. It exposes the relational annotation of the sequent calculus as recording the directed paths through the proof net constructed by the sequent proof. We give two correctness conditions: one by local rewriting in the style of Danos *contractibility* [8] and the *parsing* approach of Lafont, Guerrini and Masini [18, 14]; and a global, geometric criterion that combines Danos–Regnier switching [9] and Lamarche's *essential net* condition [19]. We introduce our proof nets with an example in Section 1.2.

We have aimed for *canonical* proof nets: those that factor out all sequent calculus permutations. To this end we have restricted ourselves to the fragment MBILL−, multiplicative bi-intuitionistic linear logic without units. MBILL with units, even though it omits negation, includes unit-only MLL, where canonical proof nets are unavailable: the proof equivalence problem, which canonical proof nets would solve efficiently, is PSPACE-complete [15].

1.1 Background and related work

In the late 1960s Lambek initiated the study of substructural logics, which restrict contraction and weakening, through category theory and with a particular focus on non-commutative variants [20]. The central point of FILL, the relation between par and linear implication, was investigated in the early 1980s by Grishin [13]. The advent of linear logic in the late 1980s [12] created an interest also in intuitionistic variants. Schellinx observed that for a multi-conclusion sequent calculus with single-conclusion $\multimap R$ rule, cut-elimination fails [22, p.555].

To obtain cut-elimination, Hyland and De Paiva formalize FILL through a sequent calculus annotated by a term calculus [17]. The terms describe natural deduction derivations whose open assumptions, identified by free variables in the terms, give a side-condition to a multi-conclusion $\multimap R$-rule similar to that of the current relational calculus. Unfortunately, as pointed out by Bierman, the term assignment introduces spurious dependencies that break cut-elimination. Three solutions to this problem were proposed: a modification of the term assignment by the first author, in private communication to Hyland and Bierman (cfr. [1]); a different term assignment using pattern matching by Bierman, [3]; and a sequent calculus with relational annotation by Braüner and De Paiva [4]. This is the calculus we adopt here, extended with subtraction. Eades and De Paiva [11] later revisited the term-annotated

calculus, with the first author's correction, to prove semantic correctness. In the late 90s the first author developed proof nets for FILL (including the MIX rule) that sequentialize into the term-annotated sequent calculus [1]. Around the same time Cockett and Seely gave a graph-like natural deduction calculus for FILL, and for the variant of BILL corresponding to the plain, un-annotated multi-conclusion sequent calculus, which collapses onto MLL [6].

Recently, Clouston, Dawson, Goré and Tiu gave annotation-free alternatives to sequent calculi, in the form of deep-inference and display calculi for BILL that enjoy cut-elimination [5].

1.2 Proof nets for MBILL− via contractibility

We will introduce our proof nets through an example. It is shown below, in two modes of representation. On the left, it is viewed as a dag-like natural deduction proof. It is built from *links*, the equivalent of a natural deduction inference, shown as solid or dashed horizontal lines connecting *premises* above to *conclusions* below. The bottom link in the example, labelled x, introduces a linear implication, and as in natural deduction, closes the corresponding assumption by a matching link also labelled x. The (green) links from negative to positive atomic formulas, a^- to a^+, are *axiom links*.

In a multiplicative linear logic such as MBILL−, each connective in the conclusion of a sequent proof is introduced once, by exactly one proof rule; that is, connectives in the conclusion are 1–1 related to inferences in the sequent proof. Proof nets are similar: connectives in *open* assumptions and conclusions correspond 1–1 to (non-axiom) links. Via this correspondence, proof nets can be represented by only the *sequent* of open assumptions and conclusions, plus the *axiom links*, connected to the atomic subformulas in the sequent. This gives the second representation below.
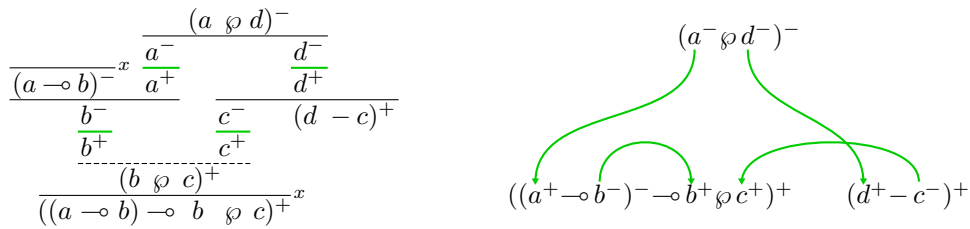
We stress that these are two different representations of one and the same graphical object, and thus the same proof net. Because the former is more explicit on logical inference, we choose it as our main representation, and as the basis of our definitions (we could have chosen either). We make axiom links explicit to emphasize the connection with the second presentation.

We may explicitly annotate formulae with their *polarity*, in the standard notion that reverses on the left of an implication. In BILL, it also reverses on the right of a subtraction. In a proof net, polarity is positive for conclusions and negative for assumptions, and indicates whether a formula is being *introduced* (+) or *eliminated* (−). An axiom link indicates a change from an elimination phase (above) to an introduction phase (below). In a sequent calculus, the negative formulae would be those in the antecedent Γ of a sequent $\Gamma \vdash \Delta$, and the positive those in the consequent Δ.

Figure 1 sequentializes the above example by contraction. It is initiated by giving an axiom for each axiom link (matched by colouring). Contraction is driven by the coloured links; in the second row, the links on a and b have contracted the −∘-elimination link between them, and the links on c and d have contracted the −-introduction link. The corresponding sequent rules are added on the right.

$$
\frac{\quad}{a \multimap b}{}^{x} \quad \frac{\dfrac{a \wp d}{a}}{a} \quad \frac{\dfrac{a \wp d}{d}}{d}
$$

$$
\frac{b}{b} \qquad \frac{c}{c} \quad d - c
$$

$$
\frac{b \wp c}{(a \multimap b) \multimap b \wp c}{}^{x}
$$

$$\overline{a \vdash a} \qquad \overline{b \vdash b} \qquad \overline{d \vdash d} \qquad \overline{c \vdash c}$$

$$
\frac{\quad}{a \multimap b}{}^{x} \quad \frac{a \wp d}{a} \quad \frac{d}{c \quad d - c}
$$

$$
\frac{b}{b \wp c}
$$

$$
\frac{b \wp c}{(a \multimap b) \multimap b \wp c}{}^{x}
$$

$$\frac{\overline{a \vdash a} \qquad \overline{b \vdash b}}{a \multimap b ,\, a \vdash b} \qquad \frac{\overline{d \vdash d} \qquad \overline{c \vdash c}}{d \vdash c ,\, d - c}$$

$$
\frac{\quad}{a \multimap b}{}^{x} \quad a \wp d
$$

$$
\frac{b \qquad c \quad d - c}{b \wp c}{}^{R}
$$

$$
\frac{b \wp c}{(a \multimap b) \multimap b \wp c}{}^{x}
$$

$$\frac{\dfrac{\overline{a \vdash a} \qquad \overline{b \vdash b}}{a \multimap b ,\, a \vdash b} \qquad \dfrac{\overline{d \vdash d} \qquad \overline{c \vdash c}}{d \vdash c ,\, d - c}}{a \multimap b ,\, a \wp d \vdash_R b ,\, c ,\, d - c}$$

$$
\frac{\quad}{a \multimap b}{}^{x} \quad a \wp d
$$

$$
\frac{b \wp c \qquad d - c}{(a \multimap b) \multimap b \wp c}{}^{x,S}
$$

$$\frac{\dfrac{\dfrac{\overline{a \vdash a} \quad \overline{b \vdash b}}{a \multimap b ,\, a \vdash b} \quad \dfrac{\overline{d \vdash d} \quad \overline{c \vdash c}}{d \vdash c ,\, d - c}}{a \multimap b ,\, a \wp d \vdash_R b ,\, c ,\, d - c}}{a \multimap b ,\, a \wp d \vdash_S b \wp c ,\, d - c}$$

$$
\frac{a \wp d}{(a \multimap b) \multimap b \wp c \quad d - c}
$$

$$\frac{\dfrac{\dfrac{\dfrac{\overline{a \vdash a} \quad \overline{b \vdash b}}{a \multimap b ,\, a \vdash b} \quad \dfrac{\overline{d \vdash d} \quad \overline{c \vdash c}}{d \vdash c ,\, d - c}}{a \multimap b ,\, a \wp d \vdash_R b ,\, c ,\, d - c}}{a \multimap b ,\, a \wp d \vdash_S b \wp c ,\, d - c}}{a \wp d \vdash (a \multimap b) \multimap b \wp c ,\, d - c}$$

$$R = \{ (a \multimap b , b) ,\ (a \wp d , b) ,\ (a \wp d , c) ,\ (a \wp d , d - c) \}$$
$$S = \{ (a \multimap b , b \wp c) ,\ (a \wp d , b \wp c) ,\ (a \wp d , d - c) \}$$

Figure 1 An example contraction and sequentialization sequence.

$$\overline{A \vdash_T A} \qquad\qquad T = \tfrac{A}{A} \qquad\qquad \frac{\Gamma \vdash_R \Delta\ A \qquad A\ \Gamma' \vdash_S \Delta'}{\Gamma\ \Gamma' \vdash_T \Delta\ \Delta'} \quad T = R \star \tfrac{A}{A} \star S$$

$$\frac{A\ B\ \Gamma \vdash_R \Delta}{A{\otimes}B\ \Gamma \vdash_T \Delta} \quad T = \tfrac{A\otimes B}{A\quad B} \star R \qquad \frac{\Gamma \vdash_R \Delta\ A \qquad \Gamma' \vdash_S \Delta'\ B}{\Gamma\ \Gamma' \vdash_T \Delta\ \Delta'\ A{\otimes}B} \quad T = (R{\cup}S) \star \tfrac{A\quad B}{A\otimes B}$$

$$\frac{C\ \Gamma \vdash_R \Delta \qquad D\ \Gamma' \vdash_S \Delta'}{C{\wp}D\ \Gamma\ \Gamma' \vdash_T \Delta\ \Delta'} \quad T = \tfrac{C\wp D}{C\quad D} \star (R{\cup}S) \qquad \frac{\Gamma \vdash_R \Delta\ C\ D}{\Gamma \vdash_T \Delta\ C{\wp}D} \quad T = R \star \tfrac{C\quad D}{C\wp D}$$

$$\frac{\Gamma \vdash_R \Delta\ A \qquad B\ \Gamma' \vdash_S \Delta'}{\Gamma\ A{\multimap}B\ \Gamma' \vdash_T \Delta\ \Delta'} \quad T = R \star \tfrac{A\multimap B\quad A}{B} \star S \qquad \frac{\Gamma\ A \vdash_R B\ \Delta}{\Gamma \vdash_T A{\multimap}B\ \Delta}\, A\!R\!\Delta \quad T = \tfrac{}{A} \star R \star \tfrac{B}{A\multimap B}$$

$$\frac{\Gamma\ D \vdash_R C\ \Delta}{\Gamma\ D{-}C \vdash_T \Delta}\, \Gamma\!R\!C \quad T = \tfrac{D-C}{D} \star R \star \tfrac{C}{} \qquad \frac{\Gamma \vdash_R \Delta\ D \qquad C\ \Gamma' \vdash_S \Delta'}{\Gamma\ \Gamma' \vdash_T \Delta\ D{-}C\ \Delta'} \quad T = R \star \tfrac{D}{C\quad D-C} \star S$$

Figure 2 Relational sequent calculus for MBILL−.

The next step contracts both active links with the $\wp$-elimination link, and introduces an explicit relation R between the premises and the conclusions of the resulting link. Its purpose is to maintain the connectedness by directed (top-down) paths through the proof net. In this case, there was no directed path from $a \multimap b$ to c or to $d - c$, and to reflect this in the link created by the contraction, the relation R connects $a \multimap b$ only to b. In the third step, the $\wp$-introduction link is contracted. It uses a dashed line because it is *switched*, and may only contract if both premises connect to the same link.

Preserving top-down connectedness is the key to showing the correctness of $\multimap$-introduction links, in the last step, which must (at least) fulfil the standard intuitionistic condition: all directed paths from the discharged assumption to an (open) conclusion must pass through the discharging $\multimap$-introduction link (see [19]). The contraction step comes with the following side-condition, analogous to that of the sequent rule: the assumption $a \multimap b$ may only be related by S to the premise of the $\multimap$-introduction link, $b \wp c$, and not to other conclusions, here $d - c$. For simplicity we omit the annotation for the final link again, as it is the full relation between premises and conclusions.

This concludes the example: the net contracts to a single link, and is thus correct.

2 MBILL−

The language of MBILL− is given by the following grammar.

$$A, B, C \ ::= \ a \ \mid \ A \otimes B \ \mid \ A \multimap B \ \mid \ A \wp B \ \mid \ A - B$$

We use $a, b, c, \dots$ to range over propositional atoms. The connectives are *tensor*, *(linear) implication*, *par*, and *subtraction*. The subformula occurrences of a formula have an implicit **polarity** $+$ or $-$, inherited from the parent formula but reversing to the left of an implication and to the right of a subtraction: $(A \multimap B)^+$ induces A^- and $(A - B)^+$ induces B^-, and similarly with $+$ and $-$ reversed.

Figure 2 gives the relational sequent calculus of Braüner and De Paiva [4], adapted for MBILL− by introducing rules for subtraction, dual to implication. A sequent is of the form $\Gamma \vdash_R \Delta$, where Γ and Δ are multisets of formulae and $R \subseteq \Gamma \times \Delta$ is a relation from Γ to Δ. (We assume that occurrences of the same formula can be distinguished, for instance by *naming* them.)

The relational annotation maintains a notion of *logical dependence* between the formulas of a sequent. Intuitively, it traces the *subformula* relation through a proof, and in addition

$$\frac{A^+ \quad B^+}{(A \otimes B)^+}\otimes I \qquad \frac{\overline{A^-}^{\,x}\atop \vdots \atop B^+}{(A \multimap B)^+}\multimap I,x \qquad \frac{A^+ \quad B^+}{(A \wp B)^+}\wp I \qquad \frac{B^+}{A^- \quad (B-A)^+}-I \qquad \frac{A^-}{A^+}\,{\color{green}ax}$$

$$\frac{(A\otimes B)^-}{A^- \quad B^-}\otimes E \qquad \frac{(A\multimap B)^- \quad A^+}{B^-}\multimap E \qquad \frac{(A\wp B)^-}{A^- \quad B^-}\wp E \qquad \frac{(B-A)^-}{B^- \atop \vdots \atop \underline{A^+}}-E,x \qquad \frac{A^+}{A^-}\,{\color{red}cut}$$

Figure 3 Links for the construction of MBILL− proof nets.

connects across axioms. An introduction rule for a linear implication $A \multimap B$ then requires that no formula other than B depends on the assumption A. This is closely related to the correctness condition of Lamarche's *essential nets* [19] for intuitionistic linear logic: all paths from A must converge on $A \multimap B$. The subtraction rule has a corresponding side-condition.

We use the following standard notation: relational composition $R;S$ of $R \subseteq \Gamma \times \Delta$ with $S \subseteq \Delta \times \Lambda$, the identity relation ID_Γ on a sequent Γ, and ARB for $(A,B) \in R$. We extend the latter by writing $\Gamma R\Delta$ if ARB for some A in Γ and B in Δ, and $\Gamma \cancel{R}\Delta$ for the negation of this proposition. We further adopt a useful notion of relational composition of Braüner and De Paiva [4]. The **star-composition** $R \star S$ of two relations $R \subseteq \Gamma \times (\Delta \cup \Delta')$ and $S \subseteq (\Delta' \cup \Delta'') \times \Lambda$, where Δ, Δ', and Δ'' are pairwise disjoint, is

$$R \star S \quad = \quad (R \cup ID_{\Delta''})\,;\,(ID_\Delta \cup S) \quad \subseteq \quad (\Gamma \cup \Delta'') \times (\Delta \cup \Lambda)$$

The above composition consists of three parts: R restricted to $\Gamma \times \Delta$, S restricted to $\Delta'' \times \Lambda$, and $R;S$ restricted to $\Gamma \times \Lambda$. It is a relational equivalent of linear distributivity [7], and a generalization of both union (if Δ' is empty) and composition (if Δ and Δ'' are empty). For ease of presentation, we write $\frac{\Gamma}{\Delta}$ for the full relation $\Gamma \times \Delta$. Note that $\overline{A}$ stands for the *empty* relation from the empty sequent to A; it is used, with $(\star)$-composition, to restrict the domain of a relation by removing A.

3 Proof nets

We shall define our proof nets for MBILL− as a graph-like natural deduction calculus. We make axioms and cuts explicit, as inference rules that only change the polarity of a formula. This gives a closer connection with sequent calculus and traditional proof nets, and simplifies the definition of contractibility. First we define the underlying graphs, or *pre-nets*; then we will introduce contractibility as a correctness condition, and define our proof nets as the pre-nets satisfying contractibility.

▶ **Definition 1** (Pre-nets). MBILL− pre-nets are built from the following notions.

▬ **Link:** a node with $n \geq 0$ **premise** ports and $m \geq 0$ **conclusion** ports labelled with formulas $A_1 \ldots A_n$ and $B_1 \ldots B_m$ and a possibly empty label ℓ. A **relational link** is labelled with a relation $R \subseteq \{A_1, \ldots, A_n\} \times \{B_1 \ldots B_m\}$. A link is drawn as follows.

$$\frac{A_1 \ \ldots \ A_n}{B_1 \ \ldots \ B_m}\ell$$

▬ **Edge:** a connection from a premise port to a conclusion port labelled with the same formula, of the same polarity.

$$\frac{A}{A}\,{}_{ax} \quad \overset{ax}{\leadsto} \quad \frac{A}{A}\,{}_{A \times A} \qquad\qquad {}_{R}\frac{\Gamma}{\Delta}\ \frac{A \quad \Gamma'}{\Delta'}\,{}_{S} \quad \overset{\star}{\leadsto}_{\Delta \cap \Gamma' = \varnothing} \quad \frac{\Gamma \quad \Gamma'}{\Delta \quad \Delta'}\,{}_{R \star S} \qquad\qquad \frac{A}{A}\,{}_{cut} \quad \overset{cut}{\leadsto} \quad \frac{A}{A}\,{}_{A \times A}$$

$$\frac{\underset{\overline{A \quad B}}{A \otimes B} \quad \Gamma}{\Delta}\,{}_{R} \quad \overset{\otimes E}{\leadsto} \quad \frac{A \otimes B \quad \Gamma}{\Delta}\,{}_{(A \otimes B \times A\ B) \star R} \qquad\qquad \frac{A \quad B}{A \otimes B} \quad \overset{\otimes I}{\leadsto} \quad \frac{A \quad B}{A \otimes B}\,{}_{A\ B \times A \otimes B}$$

$$\frac{A \multimap B \quad A}{B} \quad \overset{\multimap E}{\leadsto} \quad \frac{A \multimap B \quad A}{B}\,{}_{A \multimap B\ A \times B} \qquad\qquad \frac{\overline{A}^{\ x} \quad \Gamma}{\dfrac{B}{A \multimap B}^{\ x}}\,{}_{R} \quad \overset{\multimap I}{\underset{A\!\!\!/\Delta}{\leadsto}} \quad \frac{\Gamma}{A \multimap B \quad \Delta}\,{}_{ID_\Gamma\,;\,R \star (B \times A \multimap B)}$$

$$\frac{D \wp C}{D \quad C} \quad \overset{\wp E}{\leadsto} \quad \frac{D \wp C}{D \quad C}\,{}_{D \wp C \times D\ C} \qquad\qquad \frac{\overline{D \quad C} \quad \Gamma}{D \wp C}\,{}_{R} \quad \overset{\wp I}{\leadsto} \quad \frac{\Gamma}{D \wp C \quad \Delta}\,{}_{R \star (D\ C \times D \wp C)}$$

$$\frac{\dfrac{D - C}{D}^{\ x} \quad \Gamma}{\underset{x}{C} \quad \Delta}\,{}_{R} \quad \overset{-E}{\underset{\Gamma\!\!\!/C}{\leadsto}} \quad \frac{D - C \quad \Gamma}{\Delta}\,{}_{(D - C \times D) \star R\,;\,ID_\Delta} \qquad\qquad \frac{D}{C \quad D - C} \quad \overset{-I}{\leadsto} \quad \frac{D}{C \quad D - C}\,{}_{D \times C\ D - C}$$

Figure 4 Contraction rules.

- **Pre-net:** an acyclic directed graph $N = (V, E)$ with V a set of links as in Figure 3, and E a set of edges such that no two edges connect to the same port, satisfying the following conditions. A premise / conclusion port with no attached edge is an **open assumption / conclusion**. The $\multimap I$ / $-E$ links are in bijection with the **closed assumption / conclusion** links, defined by the variable labels x in Figure 3. A **relational pre-net** may contain also relational links.

In Figure 3, note that the illustrations for $\multimap I$ and $-E$ links each show *two* links: the $\multimap I$ link itself, plus a closed assumption link; and the $-E$ link plus a closed conclusion link.

$\dfrac{\Gamma}{\Delta}\,{}_{R}$ We abbreviate a pre-net with open assumptions Γ and open conclusions Δ by a double-lined link, as on the left. We may annotate it with a relation R that relates A in Γ to B in Δ if (and only if) there is a directed downward path from A to B.

3.1 Contractibility

Our correctness condition is in the style of Danos *contractibility* [8].[1] Contractibility for MLL proof nets is, in essence, top-down sequentialization [18, 14], starting from the axioms rather than the conclusion of a proof net. In our current natural deduction style, contraction is *inside-out*, from axioms to assumptions and conclusions. Contracting a proof net corresponds to the construction of a sequent proof or other inductive proof object. This can be made explicit by carrying the constructed object as a label on the contracting links, which we will do in Section 4.

The links of a proof net being contracted correspond to sequents of the proof being constructed. As such, we will be contracting *relational links* (see Definition 1), corresponding to relational sequents.

---

[1] The second author has also used the term *coalescence* for the generalization of contractibility that includes the additives—but as these are not currently present, we feel it is more appropriate to use the terminology that was established earlier.

$$\overline{A \vdash_T A} \quad \Rrightarrow \quad \frac{A^-}{A^+} \qquad\qquad \frac{\Gamma \vdash_R \Delta\ A \quad A\ \Gamma' \vdash_S \Delta'}{\Gamma\ \Gamma' \vdash_T \Delta\ \Delta'} \quad \Rrightarrow \quad {}_R\frac{\Gamma}{\Delta \quad A^+}$$

$$\frac{A\ B\ \Gamma \vdash_R \Delta}{A \otimes B\ \Gamma \vdash_T \Delta} \quad \Rrightarrow \quad \frac{\overline{A \otimes B}}{\overline{A}\quad \overline{B}\quad \Gamma}{}_R \qquad\qquad \frac{\Gamma \vdash_R \Delta\ A \quad \Gamma' \vdash_S \Delta'\ B}{\Gamma\ \Gamma' \vdash_T \Delta\ \Delta'\ A \otimes B} \quad \Rrightarrow \quad {}_R\frac{\Gamma}{\Delta\ \ A}\ \frac{\Gamma'}{B\ \ \Delta'}{}_S$$

$$\frac{\Gamma \vdash_R \Delta\ C\ D}{\Gamma \vdash_T \Delta\ C \wp D} \quad \Rrightarrow \quad \frac{\Gamma}{\frac{D\quad C\quad \Delta}{D \wp C}}{}_R \qquad\qquad \frac{C\ \Gamma \vdash_R \Delta \quad D\ \Gamma' \vdash_S \Delta'}{C \wp D\ \Gamma\ \Gamma' \vdash_T \Delta\ \Delta'} \quad \Rrightarrow \quad {}_R\frac{\Gamma}{\Delta}\ \frac{C \wp D}{C\quad C\quad \Gamma'}{}_S$$

$$\frac{\Gamma\ A \vdash_R B\ \Delta}{\Gamma \vdash_T A \multimap B\ \Delta}{}_{A \not\in \Delta} \quad \Rrightarrow \quad \frac{\overline{A}^{\,x}\quad \Gamma}{\frac{B\quad\quad \Delta}{A \multimap B}{}^x}{}_R \qquad\qquad \frac{\Gamma \vdash_R \Delta\ A \quad B\ \Gamma' \vdash_S \Delta'}{\Gamma\ A \multimap B\ \Gamma' \vdash_T \Delta\ \Delta'} \quad \Rrightarrow \quad {}_R\frac{\Gamma}{\Delta\quad A\quad A \multimap B}$$

$$\frac{\Gamma\ D \vdash_R C\ \Delta}{\Gamma\ D - C \vdash_T \Delta}{}_{D \not\in \Delta} \quad \Rrightarrow \quad \frac{\frac{D - C}{D}{}^x\quad \Gamma}{\frac{C}{\underline{C}{}_x}\quad \Delta}{}_R \qquad\qquad \frac{\Gamma \vdash_R \Delta\ D \quad C\ \Gamma' \vdash_S \Delta'}{\Gamma\ \Gamma' \vdash_T \Delta\ D - C\ \Delta'} \quad \Rrightarrow \quad {}_R\frac{\Gamma}{\Delta\quad D}$$

Figure 5 De-sequentialization.

▶ **Definition 2** (Contractibility). **Contraction** is the rewrite relation on relational pre-nets given by the rewrite rules in Figure 4. Contraction is successful if it terminates with a single link. A pre-net **contracts**, or is **contractible**, if it has a successful contraction path. It **strongly** contracts if every contraction path is eventually successful.

▶ **Definition 3** (Proof nets). A MBILL− **proof net** is a contractible MBILL− pre-net whose open assumptions and conclusions have negative respective positive polarity.

An example contraction sequence was given in Figure 1 in the introduction. An example of how contraction excludes incorrect nets is the following.

▶ **Example 4.** Below left is an incorrect pre-net. After several ax, $\wp E$, $\otimes I$ and $\star$ steps, we obtain the pre-net below right, where $R = \{\ (a \wp b,\, a)\,,\ (a \wp b,\, b \otimes c)\,,\ (c,\, b \otimes c)\ \}$. Because of the relation $(a \wp b,\, b \otimes c)$ this prevents further contraction: there are two potential steps, a $\multimap I$-step and a $-E$-step, and for both the side-condition is not met.

$$\frac{{}^x\frac{}{a \wp b}\quad \frac{c - (b \otimes c)}{c}{}_y}{{}^x\frac{\frac{a}{(a \wp b) \multimap a}\quad \frac{b}{b}\quad \frac{c}{b \otimes c}{}_y}{}} \qquad\qquad \frac{{}^x\frac{}{a \wp b}\quad \frac{c - (b \otimes c)}{c}{}_y}{{}^x\frac{\frac{a}{(a \wp b) \multimap a}\quad \frac{b \otimes c}{}{}_y}{}}{}_R$$

4 Sequentialization and de-sequentialization

To de-sequentialize a sequent proof to a proof net, intuitively, is to take each sequent rule, and separate the logical inference (e.g. from $A \multimap B$ and A to B) from the context (Γ and Δ). We visualize this in Figure 5, where the premises of each rule de-sequentialize to the given (double-lined) pre-nets.

▶ **Definition 5.** A sequent proof **de-sequentializes** ($\Mapsto$) to a proof net as illustrated in Figure 5.

▶ **Proposition 6.** *The de-sequentialization of a sequent proof contracts.*

Proof. By induction on the sequent proof. Following Figure 5, a de-sequentialization $\dfrac{\Gamma}{\Delta} R$ contracts to the relational link $\dfrac{\Gamma}{\Delta} R$. ◀

Sequentialization is by contraction. First, we introduce a notion of **open proof**, a sequent proof from (open) premise sequents $\vdash A$ and $B \vdash$. We abbreviate an open proof by a double line, as below left. The given open proof will result from contracting a pre-net with negative assumptions Γ^- and positive conclusions Δ^+, plus *positive* assumptions $A_1^+ \ldots A_n^+$ and *negative* conclusions $B_1^- \ldots B_m^-$, below right. The domain and range of the annotating relation of a sequent are extended to include the open premises: $R \subseteq (\Gamma\, A_1\, \ldots\, A_n) \times (\Delta\, B_1\, \ldots\, B_m)$. The relation is otherwise constructed as before.

$$\dfrac{\vdash A_1 \ldots \ \vdash A_n \quad B_1 \vdash \ \ldots \ B_m \vdash}{\Gamma \vdash_R \Delta} \qquad\qquad \dfrac{\Gamma^- \ A_1^+ \ldots A_n^+}{\Delta^+ \ B_1^- \ldots B_m^-} R$$

For sequentialization, we define a mapping from the contracting links of a proof net to sequent proofs. For a star-composition,

$$_R\dfrac{\Gamma^- \ A_1^+ \ldots A_n^+}{\Delta^+ \ B_1^- \ldots B_m^-} \ \dfrac{C^+ \ \Gamma'^- \ A_{n+1}^+ \ldots A_p^+}{\Delta'^+ \ B_{m+1}^- \ldots B_q^-}_S \qquad \overset{\star}{\leadsto} \qquad \dfrac{\Gamma^- \quad \Gamma'^- \quad A_1^+ \ldots A_p^+}{\Delta^+ \quad \Delta'^+ \quad B_1^- \ldots B_q^-}_{R \star S}$$

if the links in the redex map onto the open proofs

$$\Pi = \dfrac{\vdash A_1 \ldots \ \vdash A_n \quad B_1 \vdash \ \ldots \ B_m \vdash}{\Gamma \vdash_R \Delta\ C} \qquad \Phi = \dfrac{\vdash C \ \ \vdash A_{n+1} \ldots \ \vdash A_p \quad B_{m+1} \vdash \ \ldots \ B_q \vdash}{\Gamma' \vdash_S \Delta'}$$

then the contractum is mapped onto the open proof

$$\dfrac{\vdash A_1 \ldots \ \vdash A_p \quad B_1 \vdash \ \ldots \ B_q \vdash}{\Gamma\ \Gamma' \vdash_{R \star S} \Delta\ \Delta'}$$

obtained by replacing the open premise $\vdash C$ of Φ with the open proof Π, and adding the conclusions Γ and Δ to each inference from $\vdash C$ down to the conclusion of Φ.

To the contractum of the steps *ax*, *cut*, $\otimes I$, $\multimap E$, $\wp E$, $-I$ we assign the respective proofs:

$$\dfrac{}{A \vdash A} \qquad \dfrac{\vdash C \quad C \vdash}{\vdash} \qquad \dfrac{\vdash A \quad \vdash B}{\vdash A \otimes B} \qquad \dfrac{\vdash A \quad B \vdash}{A \multimap B \vdash} \qquad \dfrac{C \vdash \quad D \vdash}{C \wp D \vdash} \qquad \dfrac{\vdash C \quad D \vdash}{\vdash C - D}$$

To the remaining steps we assign proofs as follows, where $\Gamma = \Gamma^- \, A_1^+ \dots A_n^+$ and $\Delta = \Delta' \, B_1^- \dots B_m^-$.

$$
\frac{\overset{\text{-----}}{\dfrac{A \otimes B}{A \quad B} \quad \Gamma}}{\Delta}\,R \qquad \overset{\otimes E}{\leadsto} \qquad \frac{A \otimes B \quad \Gamma}{\Delta}\,{\scriptstyle (A \otimes B \,\times\, A\ B)\star R} \qquad\qquad \frac{\dfrac{\vdash A_1 \dots \ \vdash A_n \quad B_1 \vdash \dots \ B_m \vdash}{A \ B \ \Gamma' \vdash_R \Delta'}}{A \otimes B \ \Gamma' \vdash_T \Delta'}
$$

$$
\frac{\dfrac{\overline{A}^{\,x} \quad \Gamma}{B \quad \Delta}\,R}{A \multimap B}\,x \qquad \overset{\multimap I}{\underset{A\!\!\not R \Delta}{\leadsto}} \qquad \frac{\Gamma}{A \multimap B \quad \Delta}\,{\scriptstyle ID_\Gamma \,;\, R\star(B \,\times\, A\multimap B)} \qquad\qquad \frac{\dfrac{\vdash A_1 \dots \ \vdash A_n \quad B_1 \vdash \dots \ B_m \vdash}{\Gamma' \ A \vdash_R B \ \Delta'}}{\Gamma' \vdash_T A \multimap B \ \Delta'}\,{\scriptstyle A\!\!\not R\Delta'}
$$

$$
\frac{\dfrac{\Gamma}{D \quad C \quad \Delta}\,R}{\overset{\text{-----}}{D \wp C}} \qquad \overset{\wp I}{\leadsto} \qquad \frac{\Gamma}{D \wp C \quad \Delta}\,{\scriptstyle R\star(D\ C \,\times\, D\wp C)} \qquad\qquad \frac{\dfrac{\vdash A_1 \dots \ \vdash A_n \quad B_1 \vdash \dots \ B_m \vdash}{\Gamma' \vdash_R C \ D \ \Delta'}}{\Gamma' \vdash_T C \wp D \ \Delta'}
$$

$$
\frac{\dfrac{\overset{\text{-----}}{\dfrac{D - C}{D}}^{\,x} \quad \Gamma}{\underset{x}{C} \quad \Delta}\,R}{}\,{\scriptstyle \Gamma\!\!\not R C} \qquad \overset{-E}{\leadsto} \qquad \frac{D - C \quad \Gamma}{\Delta}\,{\scriptstyle (D-C \,\times\, D)\star R \,;\, ID_\Delta} \qquad\qquad \frac{\dfrac{\vdash A_1 \dots \ \vdash A_n \quad B_1 \vdash \dots \ B_m \vdash}{\Gamma' \ D \vdash_R C \ \Delta'}}{\Gamma' \ D{-}C \vdash_T \Delta'}\,{\scriptstyle \Gamma'\!\!\not R C}
$$

Finally, recall that a proof net has only negative assumptions and positive conclusions. If it contracts to a single link, this link maps to a regular (relational) sequent proof, without open premises.

▶ **Definition 7** (Sequentialization)**.** A proof net **sequentializes** to a proof Π if it contracts to a single link that maps onto Π.

▶ **Proposition 8.** *The de-sequentialization of a sequent proof Π sequentializes to Π.*

Proof. By induction on the sequent proof. Following Figure 5, a de-sequentialization $\dfrac{\Gamma}{\overline{\overline{\Delta}}}\,R$ of Π contracts to the relational link $\dfrac{\Gamma}{\Delta}\,R$ mapping to Π. ◀

5   A geometric characterization

In this section we give a geometric correctness condition for MBILL− proof nets, and demonstrate that a pre-net contracts if and only if it is correct. The condition has two components: a **switching** condition in the style of Danos and Regnier [9] that integrates the condition on Lamarche's *essential nets* [19], and a **bi-functionality** condition that further refines the essential net condition. We begin by giving the necessary definitions.

▶ **Definition 9** (Switching)**.** In a pre-net N:

Switched / solid. The **switched** links are $\wp I$, $\otimes E$, $\multimap I$, and $-E$; other links are **solid**. A **switched** edge is one connecting to an auxiliary port of a switched link or to a closed assumption or conclusion link; other edges are **solid**.

Targets. The **targets** of a switched link are as follows:

- the targets of a $\wp I$ or $\otimes E$ link are the two links connected by a switched edge;
- the targets of a $\multimap I$ link $A \multimap B$ are the link connected to the auxiliary port B plus all links on a directed downward path starting from the associated closed assumption link A, but not passing through $A \multimap B$;
- the targets of a $-E$ link $D - C$ are the link connected to the auxiliary port D plus all links on a directed downward path ending at the associated closed conclusion link C, but not passing through $D - C$.

Switching graph. A **switching graph** G for N is an undirected graph (V, E) whose **vertices** V are the links of N, and whose edges E connect:

- any two links connected by a solid edge in N;
- any switched link to exactly one of its targets.

Switching condition. A pre-net satisfies the **switching condition** if every switching graph is acyclic and connected.

▶ **Definition 10** (Bi-functionality)**.** A pre-net satisfies the **bi-functionality condition** if

- a directed path from a closed assumption x to an open conclusion passes through $\multimap I, x$;
- a directed path from an open assumption to a closed conclusion y passes through $-E, y$;
- a directed path from a closed assumption x to a closed conclusion y passes through $\multimap I, x$ or $-E, y$.

▶ **Remark.** Closer observation will reveal that the first two components of the bi-functionality condition are equivalent to assuming an implicit $\wp I$-link connecting all open conclusions, and a $\otimes E$-link connecting open assumptions. The third component is equivalent to considering a closed assumption x and its implication introduction link $\multimap I, x$ to be one and the same link for the purpose of the switching graph (though not for downward reachability).

▶ **Definition 11** (Geometric correctness)**.** A pre-net N is **geometrically correct** if it satisfies both the switching condition and the bi-functionality condition.

A **switching path** is an undirected path in a switching graph G, which we will indicate by ($\overset{G}{\longrightarrow}$). A single, switched edge will be written ($\overset{G}{\text{-----}}$), and we may omit the superscript if G is understood. For simplicity, we will refer to a link by its principal formula when indicating switching paths. For a link A and switched link B in a switching graph G, write $A \ll_G B$ if A is on a switching path between two targets B_1 and B_2 of B, i.e. if there is a switching path $B_1 \overset{G}{\longrightarrow} A \overset{G}{\longrightarrow} B_2$.

▶ **Definition 12.** A link A is **in scope of** a switched link B, written $A \ll B$, if $A \ll_G B$ for some G. The **scope** of a link B is the set $\{A \mid A \ll B\}$.

We take the scope relation ($\ll$) as ranging over all links, though note that for a solid link B there is never any $A \ll B$.

▶ **Lemma 13.** *In a pre-net satisfying the switching condition, ($\ll$) is a strict partial order.*

Proof. Irreflexivity: $A \not\ll A$. Immediate, since a switching path $A_1 \longrightarrow A \text{ ---- } A_2$ (with A switched to A_2) creates a cycle $A_1 \longrightarrow A \text{ ---- } A_1$ by switching A to A_1.

Transitivity: if $A \ll B \ll C$ then $A \ll C$. Let B be a switched link with jump targets B_1, B_2, and B_3, and C a switched link with targets C_1 and C_2. Let $A \ll B \ll C$ be witnessed by switching graphs G and H, so that $A \ll_G B \ll_H C$, via the following paths.

$$B_2 \overset{G}{\longrightarrow} A \overset{G}{\longrightarrow} B_3 \qquad C_1 \overset{H}{\longrightarrow} B_1 \overset{H}{\text{----}} B \overset{H}{\longrightarrow} C_2$$

We allow the possiblity that B_1 is the same as either of B_2 and B_3, as is necessarily the case for a binary switched link. First, we create a switching K which agrees with H everywhere except the links on the below path, where it agrees with those links.

$$B \text{ ---- } B_2 \overset{G}{\longrightarrow} A \overset{G}{\longrightarrow} B_3$$

Crucially, no other path in G from B may connect to the above path, and so any path in K not ending with a switched edge of B must agree with H. In particular this includes the path

B —— C_2. Moreover, in H no path from the principal port of B reaches C_1, since there is already a path C_1 —— B_1 ---- B. Then also in K no path from the principal port of B, which must all agree with H, can reach C_1. Instead, C_1 and B must then be connected as follows.

$$C_1 \xrightarrow{\;K\;} B_2 \overset{K}{\dashrightarrow} B$$

Let X be the link where this path first intersects the path $B_2 \xrightarrow{\;G\;} A \xrightarrow{\;G\;} B_3$, where K agrees with G; without loss of generality, assume thats X comes before A. This gives the following.

$$C_1 \xrightarrow{\;K\;} X \xrightarrow{\;K\;} B_2 \qquad B_2 \xrightarrow{\;K\;} X \xrightarrow{\;K\;} A \xrightarrow{\;K\;} B_3$$

Switching B to B_3 we have the following path.

$$C_1 \xrightarrow{\;K\;} X \xrightarrow{\;K\;} A \xrightarrow{\;K\;} B_3 \text{ ---- } B \xrightarrow{\;K\;} C_2$$

Then $A \ll C$, as required. ◀

Our notion of scope is related to the first author's notion of *loop* for MLL nets with Mix [1]. It is further closely related to the De Naurois–Mogbil correctness condition [10]. This uses the relation $(\ll_G)$, over a fixed switching graph G. Unlike $(\ll)$ the relation $(\ll_G)$ is not necessarily transitive. We write $(\ll_G^*)$ for the transitive closure and $(\ll_G^n)$ for the n-fold relational composition,

$$A_0 \ll_G^n A_n \quad = \quad A_0 \ll_G A_1 \ll_G \cdots \ll_G A_n \; .$$

▶ **Proposition 14.** *In a pre-net satisfying the switching condition, $A \ll_G^* B$ if and only if $A \ll B$.*

Proof. From left to right, $A \ll_G B$ implies $A \ll B$, and $(\ll)$ is transitive. From right to left, we proceed by induction on the distance between A and B in $(\ll)$. First consider the case where A and B are immediate neighbours (distance 1), i.e. there is no C such that $A \ll C \ll B$. Then there is a path between the premises of B that does not contain any switched links. Whichever way G switches on B, we have $A \ll_G B$. In the case where there is a C such that $A \ll C \ll B$, by induction we have $A \ll_G^* C$ and $C \ll_G^* B$, and hence $A \ll_G^* B$. ◀

The scope of a link A includes exactly those links that must be contracted before A can be contracted itself. (We will use this to prove that a correct pre-net contracts, by demonstrating that any link that is minimal in $(\ll)$ may be contracted, as part of the proof of Theorem 16 below.) The scope of A then corresponds to the smallest open subproof of A in any sequentialization. In this way, the notion of scope is also closely related to the standard notion of **kingdom** [2]: the kingdom kA of a subformula A corresponds to the smallest *subproof* of A in any sequentialization.

For an MLL proof net, the kingdom kA is the smallest subgraph such that $A \in kA$ and:
1. if $B \in kA$ and B is in an axiom link with $B^\perp$, then $B^\perp \in kA$;
2. if $B \otimes C \in kA$ then $B \in kA$ and $C \in kA$;
3. If $B \wp C \in kA$ then kA includes the scope of $B \wp C$: if $D \ll B \wp C$ then $D \in kA$.
Observe that (2) corresponds to the fact that a *subproof* containing $B \otimes C$ must contain also subproofs for B and for C; however, an *open subproof* need not. Because scope is transitive, and because it does not need to be closed under (2) like kingdoms, we may avoid an inductive definition. Interestingly, this implies that (smallest) *open* subproofs are a *geometric* concept, not an inductive one.

We will now show that contractibility and geometric correctness are equivalent conditions. First, we establish that if N contracts to M, then if either of N and M is geometrically correct, both are. This is a straightforward induction on the contraction sequence.

$$(a) \quad {}_R\frac{\Gamma}{\Delta \quad A} \quad \underline{\frac{\Gamma'}{\Delta'}}\, S \quad \overset{\star}{\leadsto} \quad \frac{\Gamma}{\Delta} \quad \frac{\Gamma'}{\Delta'}\, {}_{R\star S}$$

$$(b) \quad \frac{\dfrac{\overline{A}\,{}^x \quad \Gamma}{B}\, {}_R}{A \multimap B}\, {}^x_{A\not{R}\Delta} \quad \overset{\multimap I}{\leadsto} \quad \frac{\Gamma}{A \multimap B \quad \Delta}\, {}_{ID_\Gamma\,;\,R\star(B\,\times\,A\multimap B)}$$

▶ **Lemma 15.** *Contraction preserves and reflects geometric correctness.*

Proof. We will treat the star-contraction rule (a) and the contraction rule for linear implication (b); the other rules are similar, or trivial.

Let $N \leadsto M$ by a $\star$-step. The composition $R\star S$ ensures that directed paths are maintained through the contraction step. It follows that the targets of any $\multimap I$ or $-E$ link are the same in both N and M, save that if one of both contracted links in N is a target then the resulting link in M is a target, and vice versa. This leaves the geometry of the switching graphs in N and M unchanged.

Next, let $N \leadsto M$ by a $\multimap I$-step. Because of the side-condition $A\not{R}\Delta$, the only target of the link $A \multimap B$ is the contraction link R. It follows that there is a one-to-one correspondence between switching graphs in N and in M, preserving their geometry. ◀

▶ **Theorem 16.** *A pre-net N contracts if and only if it is geometrically correct.*

Proof. From left to right, assume that N contracts. The end result, a single contracted link, is geometrically correct. Since contraction reflects geometric correctness, by Lemma 15, by induction on the contraction sequence N is geometrically correct.

From right to left, it must be shown that if N is geometrically correct, a contraction step applies. As contraction preserves geometric correctness (Lemma 15), it then follows that N contracts, by induction on its size.

Contraction steps on solid links have no side conditions, and the star-contraction rule (a) applies to any adjacent relational links. Applying these steps first, we may assume that N consists solely of relational links separated by switched links. Consider a switched link that is minimal in $(\ll)$. We will treat the case of a $\multimap I$ link $A\multimap B$ and show that a $\multimap I$-step (b) applies; the other three cases are similar.

Let X be the link connected to the port A of the closed assumption of $A\multimap B$, and Y the link connected to the auxiliary port B of the link $A\multimap B$. In any switching graph G the links X and Y must be connected, and since both are targets of $A\multimap B$, they cannot be connected through its principal port, as this would violate irreflexivity of $(\ll)$. Because $A\multimap B$ is minimal in $(\ll)$ there can be no switched link on the switching path $X \relbar\joinrel\relbar Y$, and since relational links are not adjacent (they would have been contracted), there can be only one. Then $X = Y$ is the unique relational link to which both ports A and B connect, as required by the $\multimap I$ contraction step (b).

Finally, we show that the side condition $A\not{R}\Delta$ is satisfied. Suppose there is a port D in Δ such that ARD. By the bi-functionality condition D cannot be an open conclusion, and cannot connect to a closed one. The link L connected at D must then be a switched link (since adjacent relational links were assumed to have been contracted). Note that L is a target of $A\multimap B$. If L is a $\otimes E$, $\multimap I$, or $\wp I$ link, also at least one link connected at an auxiliary

$$\frac{\dfrac{A \quad B}{A \otimes B}}{\dfrac{A \otimes B}{A \quad B}} \quad [\otimes] \quad \dfrac{A}{A} \quad \dfrac{B}{B} \qquad \dfrac{\dfrac{\overline{A}^{\,x}}{\vdots}}{\dfrac{\dfrac{B}{A \multimap B}^{\,x}}{\dfrac{A \multimap B \quad A}{B}}} \quad [\multimap] \quad \dfrac{\dfrac{A}{A}}{\dfrac{\vdots}{\dfrac{B}{B}}} \qquad \dfrac{\dfrac{A}{A}^{\,ax}}{A}^{\,cut} \quad [R] \quad A$$

$$\frac{\dfrac{C \quad D}{C \wp D}}{\dfrac{C \wp D}{C \quad D}} \quad [\wp] \quad \dfrac{C}{C} \quad \dfrac{D}{D} \qquad \dfrac{\dfrac{C \quad \dfrac{D}{D - C}}{\dfrac{D - C}{D}}^{\,x}}{\dfrac{\vdots}{\underline{C}_{\,x}}} \quad [-] \quad \dfrac{\dfrac{D}{D}}{\dfrac{\vdots}{\dfrac{C}{C}}} \qquad \dfrac{\dfrac{C}{C}^{\,cut}}{C}^{\,ax} \quad [L] \quad C$$
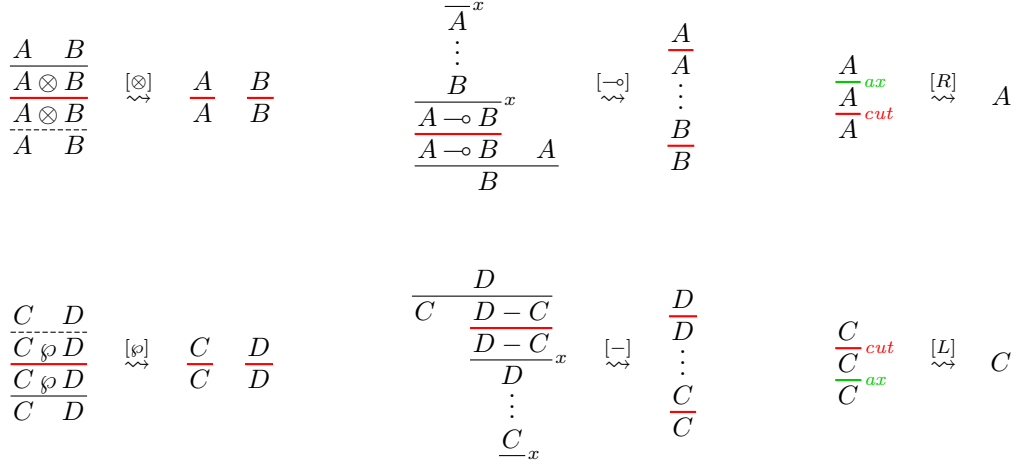
■ **Figure 6** Proof net normalization rules.

port of L (possibly X) is a target of $A \multimap B$. This would mean $L \ll (A \multimap B)$, contradicting the assumption that $A \multimap B$ was minimal. It follows that $A \not{R} \Delta$, and a $\multimap I$-contraction step applies to $A \multimap B$. ◀

To be effective, it is crucial to have *strong* contractibility, where *any* contraction path (eventually) terminates with a single link. If only some paths would eventually be successful, an algorithm for correctness would need to backtrack (or have a guaranteed strategy). Instead, we should be able to use any contraction sequence, without the chance of failure. This is established by the following theorem.

▶ **Theorem 17** (Strong contractibility). *MBILL− proof nets are strongly contractible.*

Proof. Since proof nets are correct (Theorem 16), and contraction preserves correctness (Lemma 15), any contraction step yields a correct proof net, which must then contract (Theorem 16). ◀

6   Normalization

We give proof reduction as a graph-rewrite relation on pre-nets. There are six reduction steps, one for each connective and two for axioms, given in Figure 6. Since proof nets strictly reduce in size, termination is immediate. So is confluence: the only redexes that may overlap are $[L]$ and $[R]$, but this critical pair converges trivially. A pre-net is in **normal form** if it has no cuts, and in **expanded normal form** if in addition the formulas of axiom links are atomic. The unique expanded normal form of a net N is denoted $N{\downarrow}$. The example in the introduction is in expanded normal form.

▶ **Theorem 18** (Normalization preserves correctness). *A proof net reduces to a proof net.*

Proof. By inspection of the normalization steps, geometric correctness is preserved. ◀

6.1   One-step composition

Proof nets in expanded normal form have a compact alternative representation. In a purely multiplicative logic such as MBILL−, a proof (or proof net) has exactly one rule (or link)

for every connective in the conclusion sequent. Identifying links with connectives, we can display a proof net by drawing its open assumptions (above) and conclusions (below), and connecting these with the axiom links. An example was given in the introduction; here is another.
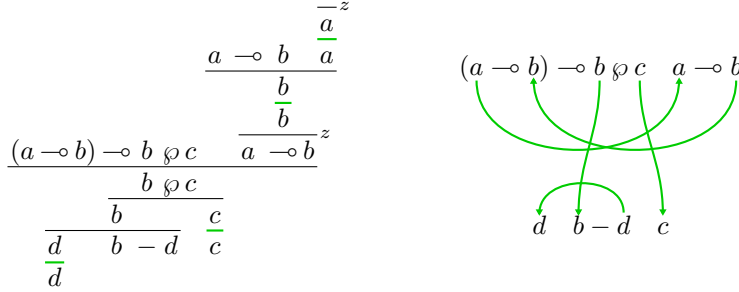
We will formalize such proof nets as the **compact form** of a net in expanded normal form. As in classical and intuitionistic MLL [16], composition of compact forms in MBILL− is particularly nice: it is path-composition along the axiom links of both nets, as connected through the formula along which they are composed. This is demonstrated below. On the left are the net from the introduction, in blue, and that from above in red (with the assumption $a \multimap b$ re-positioned on the left), with their common open conclusion and assumption superimposed. Composing these nets along that common formula gives the net below right.
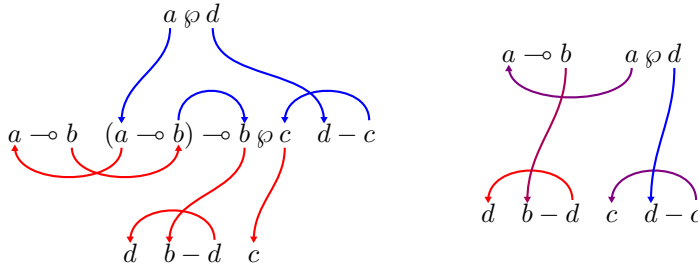
We will formalize this concisely, as follows.

▶ **Definition 19.** The **compact form** $\lfloor N \rfloor = \Lambda \colon \Gamma \vdash \Delta$ of a pre-net N in expanded normal form consists of the open assumptions Γ, the open conclusions Δ, and the axiom links Λ of N.

Given two compact forms $\lfloor M \rfloor = \Lambda_M \colon \Gamma_M \vdash \Delta_M\, A^+$ and $\lfloor K \rfloor = \Lambda_K \colon A^-\, \Gamma_K \vdash \Delta_K$, define their **composition along** A as $\Lambda \colon \Gamma_M\, \Gamma_K \vdash \Delta_M\, \Delta_K$ where Λ consists of all maximal paths in the undirected graph formed by Λ_M, Λ_K, and connecting corresponding atoms in A^+ and A^-. Correspondingly for (non-compact) pre-nets, the **cut-composition along** A of pre-nets M with open conclusion A^+ and K with open assumption A^-, is the (disjoint) union of both graphs together with a cut-link with premise A^+ and conclusion A^-.

▶ **Theorem 20.** *If N is the cut-composition along A of proof nets M and K in expanded normal form, then $\lfloor N{\downarrow} \rfloor$ is the composition along A of $\lfloor M \rfloor$ and $\lfloor K \rfloor$.*

Proof. By induction on the cut-formula. ◀

References

1 Gianluigi Bellin. Subnets of proof-nets in multiplicative linear logic with MIX. *Mathematical Structures in Computer Science*, 7(6):663–669, 1997.

2 Gianluigi Bellin and Jacques van de Wiele. Subnets of proof-nets in MLL⁻. In *Advances in Linear Logic*, pages 249–270, 1995.

3 G.M. Bierman. A note on full intuitionistic linear logic. *Annals of Pure and Applied Logic*, 79(3):281–287, 1996.

4 Torben Braüner and Valeria de Paiva. A formulation of linear logic based on dependency-relations. In *11th International Workshop on Computer Science Logic (CSL)*, 1997.

5 Ranald Clouston, Jeremy Dawson, Rajeev Gore, and Alwen Tiu. Annotation-free sequent calculi for full intuitionistic linear logic. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 23. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.

6 Robin Cockett and Robert Seely. Proof theory for full intuitionistic linear logic, bilinear logic, and MIX categories. *Theory and Applications of Categories*, 3(5):85–131, 1997.

7 Robin Cockett and Robert Seely. Weakly distributive categories. *Journal of Pure and Applied Algebra*, 114:133–173, 1997.

8 Vincent Danos. *La Logique Linéaire appliquée à l'étude de divers processus de normalisation (principalement du Lambda-calcul)*. PhD thesis, Université Paris 7, 1990.

9 Vincent Danos and Laurent Regnier. The structure of multiplicatives. *Archive for Mathematical Logic*, 28:181–203, 1989.

10 Paulin Jacobé De Naurois and Virgile Mogbil. Correctness of linear logic proof structures is NL-complete. *Theoretical Computer Science*, 412(20):1941–1957, 2011.

11 Harley Eades and Valeria de Paiva. Multiple conclusion linear logic: Cut elimination and more. In *International Symposium on Logical Foundations of Computer Science (LFCS)*, pages 90–105, 2016.

12 Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

13 V. N. Grishin. On a generalization of the ajdukiewicz–lambek system. In *Studies in Non-Classical Logics and Formal Systems*, pages 315–343. Nauka, Moscow, 1983.

14 Stefano Guerrini and Andrea Masini. Parsing MELL proof nets. *Theoretical Computer Science*, 254(1-2):317–335, 2001.

15 Willem Heijltjes and Robin Houston. Proof equivalence in MLL is PSPACE-complete. *Logical Methods in Computer Science*, 12(1), 2016.

16 Dominic J.D. Hughes. Simple free star-autonomous categories and full coherence. *Journal of Pure and Applied Algebra*, 216(11):2386–2410, 2012.

17 Martin Hyland and Valeria de Paiva. Full intuitionistic linear logic. *Annals of Pure and Applied Logic*, 64(3):273–291, 1993.

18 Yves Lafont. From proof nets to interaction nets. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, volume 222 of *London Mathematical Society Lecture Notes*, pages 225–247. Cambridge University Press, 1995.

19 François Lamarche. Proof nets for intuitionistic linear logic: Essential nets. Research report <inria-00347336>, INRIA, 2008.

20 Joachim Lambek. Deductive systems and categories II. *Lecture Notes in Mathematics*, 86:76–122, 1969.

21 Andrzej S. Murawski and C.-H. Luke Ong. Exhausting strategies, joker games and full completeness for IMLL with unit. *Theoretical Computer Science*, 294(1):269–305, 2003.

22 Harold Schellinx. Some syntactical observations on linear logic. *J. Logic and Computation*, 1(4):537–559, 1991.

A   Relations with existing syntax.

Lamarche [19] (see also Murawski and Ong [21]) developed a system of *essential nets* for ILL where nets are polarized, edges are directed and the polarization of links reflects the structure of ILL sequent calculus inferences. Notice that a $\wp^-$ link is *not switched* and $\wp^+$ links have a canonical *right* switch. The links of polarized classical MLL− formuas correspond to the intuitionistic ILL− inferences in red.
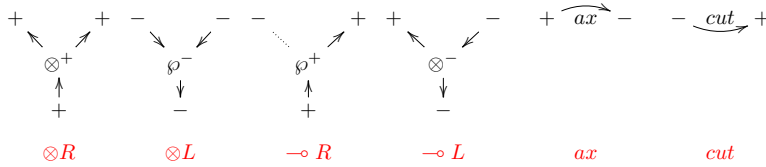
▶ **Definition 21.** An **essential net** $\mathcal{E}$ is a structure satisfying the following conditions:
1. *(acyclicity)* there is no cycle of *directed edges* in $\mathcal{E}$;
2. *(functionality of implications) for every $\wp_+$ link with premises A^- and B^+, every directed path from (the only positive) conclusion of $\mathcal{E}$ to A^- passes through B^+.*

Lamarche proves that every correct proof net can be sequentialized into an ILL sequent derivation.

▶ **Example 22.** Essential net for $q \otimes (q \multimap r) \vdash (r \multimap p) \multimap p$, where $X = q \otimes (q \multimap r)$ and $Y = (r \multimap p) \multimap p$.
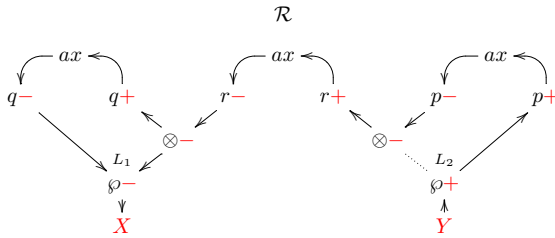
In order to extend the above representation to FILL− and BILL− we may add links for intuitionistic *par* and *subtraction*, below left. However, in this extension it is no longer possible to verify the *acyclicity* condition on *directed* paths. There is no directed cycle in the pre-net below right:
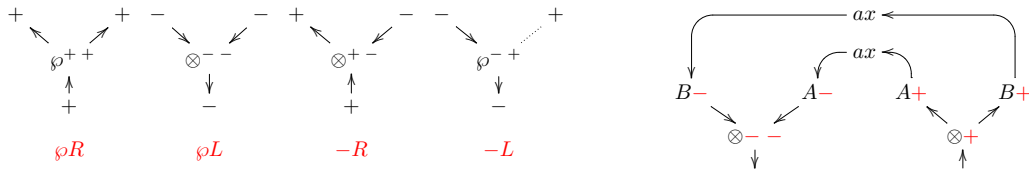
A solution is *first* test the MLL− acyclicity and connectedness condition of *undirected* DR-graphs with switchings on *par-like* links, namely, links representing MBILL− $\otimes L$, $\wp R$ (for $\multimap R$ and $-L$ the switching is canonical), and then test a specific correctness condition, the *bifunctionality condition on $\multimap R$ and $-L$*.

The first author [1] sequentializes proof nets for FILL into Hyland and De Paiva's labelled sequent calculus.

▶ **Definition 23.** A proof net $\mathcal{R}$ for FILL- is a polarized MLL- structure satisfying the following conditions:

1. (*DR condition*) for every switching s, $s\mathcal{R}$ is acyclic and connected;
2. (*functionality of implications*) *for every* $\wp_+$ *link with premises* A^- *and* B^+, *and conclusion* $(A \wp B)^+$ *every directed path from any* *positive* *conclusion* X^+ *of* $\mathcal{R}$ *to* A^- *passes through* $(A \wp B)^+$.

To prove sequentialization the following lemma is needed:

Lemma. *Let* $\mathcal{D}$ *be a labelled sequent calculus derivation of* S *and let* $\mathcal{D}^-$ *be the polarized proof net resulting from de-sequentializing* $\mathcal{D}^-$. *Then* $x : A$ *occurs in* $t : B$ *in some sequent of* $\mathcal{D}$ *iff there is a directed path from* $(B')^+$ *to* $(A')^-$ *in* $\mathcal{D}^-$, *where* $(B')^+$ *and* $(A')^-$ *are the translations of* B *and* A *in polarized MLL.*

▶ **Example 24.**
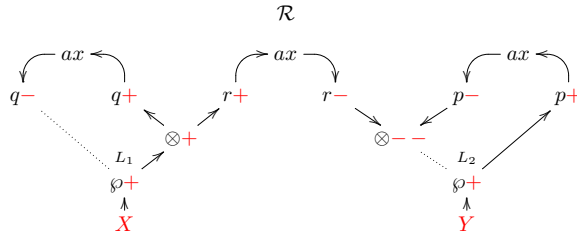
Here $X = q \multimap (q \otimes r)$, $Y = (r \wp p) \multimap p$ and there is a directed path from X to the premise $r \wp p$ of Y against the functionality of implication. In the following sequent derivation

$$
\cfrac{
\cfrac{
\cfrac{
\cfrac{y : q \vdash y : q \qquad z : r \vdash z : r}{y : q, z : r \vdash y \otimes z : q \otimes r} \qquad x : p \vdash x : p
}{v : r \wp p, y : q \vdash \mathtt{let}\ v\ \mathtt{be}\ z^r\ -\ \mathtt{in}\ y \otimes z : q \otimes r, \mathtt{let}\ v\ \mathtt{be}\ -x\ \mathtt{in}\ x : p}
}{v : r \wp p \vdash \lambda y.\mathtt{let}\ v\ \mathtt{be}\ z^r\ -\ \mathtt{in}\ y \otimes z : q \multimap q \otimes r, \mathtt{let}\ v\ \mathtt{be} - x^p\ \mathtt{in}\ x : p} \multimap R
}{\vdash \lambda y.\mathtt{let}\ v\ \mathtt{be}\ z^r\ -\ \mathtt{in}\ y \otimes z : q \multimap q \otimes r, \lambda v.\mathtt{let}\ v\ \mathtt{be} - x^p\ \mathtt{in}\ x : (r \wp p) \multimap p} \multimap R
$$

the last inference $\multimap R$ is incorrect because v still occurs free in the succedent.

Counting Environments and Closures

Maciej Bendkowski[1]
Jagiellonian University,
Faculty of Mathematics and Computer Science,
Theoretical Computer Science Department,
ul. Prof. Łojasiewicza 6, 30–348 Kraków, Poland
maciej.bendkowski@tcs.uj.edu.pl

Pierre Lescanne
University of Lyon, École normale supérieure de Lyon,
LIP (UMR 5668 CNRS ENS Lyon UCBL),
46 allée d'Italie, 69364 Lyon, France
pierre.lescanne@ens-lyon.fr

Abstract

Environments and closures are two of the main ingredients of evaluation in lambda-calculus. A closure is a pair consisting of a lambda-term and an environment, whereas an environment is a list of lambda-terms assigned to free variables. In this paper we investigate some dynamic aspects of evaluation in lambda-calculus considering the quantitative, combinatorial properties of environments and closures. Focusing on two classes of environments and closures, namely the so-called plain and closed ones, we consider the problem of their asymptotic counting and effective random generation. We provide an asymptotic approximation of the number of both plain environments and closures of size n. Using the associated generating functions, we construct effective samplers for both classes of combinatorial structures. Finally, we discuss the related problem of asymptotic counting and random generation of closed environments and closures.

1 Introduction

Though, traditionally, computational complexity is investigated in the context of Turing machines since their initial development, evaluation complexity in various term rewriting systems, such as λ-calculus or combinatory logic, attracts increasing attention only quite recently. For instance, let us mention the worst-case analysis of evaluation, based on the invariance of unitary cost models [26, 3, 1] or transformation techniques proving termination of term rewriting systems [2].

Much like in classic computational complexity, the corresponding average-case analysis of evaluation in term rewriting systems follows a different, more combinatorial and quantitative approach, compared to its worst-case variant. In [10, 11] Choppy, Kaplan and Soria propose an average-case complexity analysis of normalisation in a general class of term rewriting systems using generating functions, in particular techniques from analytic combinatorics [19].

Following a somewhat similar path, Bendkwoski, Grygiel and Zaionc investigated later the asymptotic properties of normal-order reduction in combinatory logic, in particular the normalisation cost of large random combinators [7, 4]. Alas, normalisation in λ-calculus has not yet been studied in such a combinatorial context. Nonetheless, static, quantitative properties of λ-terms, form an active stream of recent research. Let us mention, non-exhaustively, investigations into the asymptotic properties of large random λ-terms [15, 6] or their effective counting and random generation ensuring a uniform distribution among terms with equal size [8, 23, 22, 9].

In the current paper, we take a step towards the average-case analysis of reduction complexity in λ-calculus. Specifically, we offer a quantitative analysis of environments and closures — two types of structures frequently present at the core of abstract machines modelling λ-term evaluation, such as for instance the Krivine or U- machine [13, 28]. In Section 3 we discuss the combinatorial representation of environments and closures, in particular the associated de Bruijn notation. In Section 4 we list the analytic combinatorics tools required for our analysis. Next, in Section 5 and Section 6 we conduct our quantitative investigation into so-called plain and closed environments and closures, respectively, subsequently concluding the paper in Section 7.

2   A combinatoric approach to higher order rewriting systems

As said in the introduction, viewing the λ-calculus from the perspective of counting is new, especially in the scientific community of structures for computation and deduction and requires motivation to be detailed.

First, clearly a new perspective on λ-calculus enlightens the semantics and opens new directions, especially by adding a touch of efficiency and a discussion on how the size of structures with binders (like λ-terms) can be measured. However, despite advanced mathematical techniques are used, the goal is more practical and connected to operational semantics and implementation. Counting allows assigning a precise measure on how a specific algorithm performs. In [24][2] Knuth calls analysis of Type A an *analysis of a particular algorithm* and shows how important it is in computer science. He adds (p. 3): "Complexity analysis provides an interesting way to sharpen our tools for the more routine problems we face from day to day."

Furthermore, a notion of probabilistic distribution as used in the average-case analysis of algorithm, after Sedgewick and Flajolet [35], is deduced. In particular a notion of uniform distribution is inferred in order to evaluate the *average case efficiency* of algorithms w.r.t. this distribution. In this paper, the algorithms the authors have in mind are the several reduction machines for the λ-calculus, especially the Krivine machine and the U-machine, for which analyses of Type A and more specifically average case analyses are expected to be built. Another application is *random generation* of terms and several kinds of structures for computation and deduction as used for instance in QuickCheck [12]. A fully and mathematically justified random generator can only be built using the kind of tools developed in this paper.

But average case analysis based on uniform distribution is not the only one. The so-called *smoothed analysis of algorithms* [36] is another family of tools which is based on measures of size. Here the distribution is no more uniform and this method has promising applications, hopefully in structures for computation and deduction.

---

[2] This paper is part of the book "Selected Papers on Analysis of Algorithms" [25] dedicated to Professor N. G. de Bruijn.
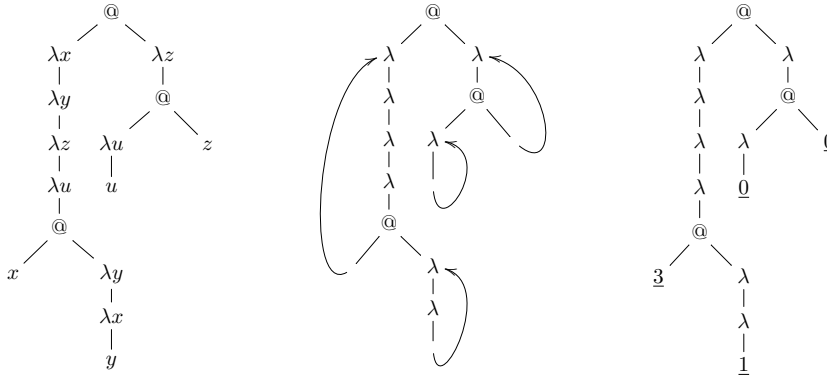
Figure 1 Three representations of the λ-term $T = (\lambda xyzu.x(\lambda yx.y))\ (\lambda z.(\lambda u.u)z)$.

3 Environments and closures

In this section we outline the de Bruijn notation and related concepts deriving from λ-calculus variants with explicit substitutions used in the subsequent sections.

3.1 De Bruijn notation

Though the classic variable notation for λ-terms is elegant and concise, it poses considerable implementation issues, especially in the context of substitution resolution and potential name clashes. In order to accommodate these problems, de Bruijn proposed an alternative name-free notation for λ-terms [16]. In this notation, each variable x is replaced by an appropriate non-negative integer $\underline{n}$ (so-called *index*) intended to encode the distance between x and its binding abstraction. Specifically, if x is bound to the $(n+1)$st abstraction on its unique path to the term root in the associated λ-tree, then x is replaced by the index $\underline{n}$. In this manner, each closed λ-term in the classic variable notation is representable in the de Bruijn notation.

▶ **Example 1.** Consider the λ-term $T = (\lambda xyzu.x(\lambda yx.y))\ (\lambda z.(\lambda u.u)z)$. Figure 1 depicts three different representations of T as tree-like structures. The first one uses explicit variables, the second one uses back pointers to represent the bound variables, whereas the third one uses de Bruijn indices.

In order to represent free occurrences of variables, one uses indices of values exceeding the number of abstractions crossed on respective paths to the term root. For instance, $\lambda x.yz$ can be represented as $\lambda \underline{12}$ since $\underline{1}$ and $\underline{2}$ correspond to two different variable occurrences.

Recall that in the classic variable notation a λ-term M is said to be *closed* if each of its variables is bound. In the de Bruijn notation, it means that for each index occurrence $\underline{n}$ in M one finds at least $n+1$ abstractions on the unique path from $\underline{n}$ to the term root of M. If a λ-term is not closed, it is said to be *open*. If heading M with m abstractions turns it into a closed λ-term, then M is said to be *m-open*. In particular, closed λ-terms are 0-open.

▶ **Example 2.** Note that $\lambda\lambda\lambda\lambda(\underline{3}(\lambda\lambda\underline{1}))\ (\lambda(\lambda\underline{0})\underline{0})$ is closed. The λ-term $\underline{3}(\lambda\lambda\underline{1})$ is 4-open, however it is not 3-open. Indeed, $\lambda\lambda\lambda(\underline{3}(\lambda\lambda\underline{1}))$ is 1-open instead of being closed. Similarly, $\lambda(\underline{3}(\lambda\lambda\underline{1}))$ is 3-open, however it is not 2-open.

Certainly, the set $\mathcal{L}_m$ of m-open terms is a subset of the set of $(m+1)$-open terms. In other words, if M is m-open, it is also $(m+1)$-open. The set of all λ-terms is called the set of *plain* terms. It is the union of the sets of m-open terms and is denoted as $\mathcal{L}_\infty$. Hence,

$$\mathcal{L}_0 \subseteq \mathcal{L}_1 \subseteq \cdots \subseteq \mathcal{L}_m \subseteq \mathcal{L}_{m+1} \cdots \subseteq \bigcup_{i=0}^{\infty} \mathcal{L}_i = \mathcal{L}_\infty \,. \tag{1}$$

Let us note that de Bruijn's name-free representation of λ-terms exhibits an important combinatorial benefit. Specifically, each λ-term in the de Bruijn notation represents an entire α-equivalence class of λ-terms in the classical variable notation. Indeed, two variable occurrences bound by the same abstraction are assigned the same de Bruijn index. In consequence, counting λ-terms in the de Bruijn notation we are, in fact, counting entire α-equivalence classes instead of their inhabitants.

3.2   Closures and β-reduction

Recall that the main rewriting rule of λ-calculus is β-*reduction*, see, e.g. [14]

$$(\beta) \quad (\lambda M)\, N \quad \rightarrow \quad M\{\underline{0} \leftarrow N\} \tag{2}$$

where the operation $\{\underline{n} \leftarrow M\}$, i.e. substitution of λ-terms for de Bruijn indices, is defined inductively as follows:

$$(M\,N)\{\underline{n} \leftarrow P\} = M\{\underline{n} \leftarrow P\}\,N\{\underline{n} \leftarrow P\}$$
$$(\lambda M)\{\underline{n} \leftarrow P\} = \lambda(M\{\underline{(n+1)} \leftarrow P\})$$
$$\underline{m}\{\underline{n} \leftarrow P\} = \begin{cases} \underline{m-1} & \text{if } m > n \\ \tau_0^n(P) & \text{if } m = n \\ \underline{m} & \text{if } m < n\,. \end{cases} \tag{3}$$

Th first rule distributes the substitution in an application, the second rule pushes a substitution under an abstraction and the third rule tells how a substitution acts when the term is an index. $\tau_0^n(P)$ tells how to update the indices of a term which is substituted for an index. The operation $\tau_i^n(M)$ is defined by induction on M as

$$\tau_i^n(M\,N) = \tau_i^n(M)\,\tau_i^n(N)$$
$$\tau_i^n(\lambda M) = \lambda(\tau_{i+1}^n(M))$$
$$\tau_i^n(\underline{m}) = \begin{cases} \underline{m+n-1} & \text{if } m > i \\ \underline{m} & \text{if } m \leq i\,. \end{cases} \tag{4}$$

A λ-term in the form of $(\lambda M)\,N$ is called a β-*redex* (or simply a *redex*). Lambda terms not containing β-redexes as subterms, are called $(\beta$-)normal forms. The computational process of rewriting (reducing) a λ-term to its β-normal form by successive elimination of β-redexes is called *normalisation*. There exists an abundant literature on normalisation in λ-calculus; let us mention, not exhaustively [27, 33, 29, 13, 30].

One of the central concepts present in various formalisms dealing with normalisation in λ-calculus are environments and closures. An *environment* is a list of values meant to be assigned to indices $\underline{0}, \underline{1}, \underline{2}, \ldots, \underline{m-1}$ of an m-open λ-term. A *closure*, on the other hand, is a couple consisting of a λ-term and an environment. Such couples are meant to represent closed, not yet fully evaluated, λ-terms. For instance, the closure $\langle M, \square \rangle$ consists of the

λ-term M evaluated in the context of an empty environment, denoted as $\square$, and represents simply M. The closure $\langle \underline{1}\,\underline{0}, \langle \lambda \underline{0}, \square \rangle : \langle \lambda \lambda \underline{0}, \square \rangle : \square \rangle$ represents the λ-term $(\underline{1}\,\underline{0})$ evaluated in the context of an environment $\langle \lambda \underline{0}, \square \rangle : \langle \lambda \lambda \underline{0}, \square \rangle : \square$. Here, intuitively, the index $\underline{1}$ is receiving the value $\lambda \underline{0}$ whereas the index $\underline{0}$ is being assigned $\lambda \lambda \underline{0}$. Finally, $\lambda \underline{0}$ is applied to $\lambda \lambda \underline{0}$. And so, reducing the closure $\langle \underline{1}\,\underline{0}, \langle \lambda \underline{0}, \square \rangle : \langle \lambda \lambda \underline{0}, \square \rangle : \square \rangle : \square$, for instance using a Krivine abstract machine [13], we obtain $\lambda \lambda \underline{0}$.

Let us notice that following the outlined description of environments and closures, we can provide a formal combinatorial specification for both using the following mutually recursive definitions:

$$\begin{aligned}
\mathcal{C}los &::= \langle \Lambda, \mathcal{E}nv \rangle \\
\mathcal{E}nv &::= \square \mid \mathcal{C}los : \mathcal{E}nv
\end{aligned} \tag{5}$$

In the above specification, Λ denotes the set of all plain λ-terms. Moreover, we introduce two binary operators "$\langle\_,\_\rangle$", i.e. the *coupling* operator, and "$:$", i.e. the *cons* operator, heading its left-hand side on the right-hand list. When applied to a λ-term and an environment, the coupling operator constructs a new closure. In other words, a *closure* is a couple of a λ-term and an environment whereas an environment is a list of closures, representing a list of assignments to free occurrences of de Bruijn indices.

Such a combinatorial specification for closures and environments plays an important rôle as it allows us to investigate, using methods of analytic combinatorics, the quantitative properties of both closures and environments.

4 Analytic tools

In the following section we briefly[3] outline the main techniques and notions from the theory of generating functions and singularity analysis. We refer the curious reader to [19, 37, 21] for a thorough introduction.

Let $(f_n)_n$ be a sequence of non-negative integers. Then, the *generating function $F(z)$* associated with $(f_n)_n$ is the formal power series $F(z) = \sum_{n \geq 0} f_n z^n$. Following standard notational conventions, we use $[z^n]F(z)$ to denote the coefficient standing by z^n in the power series expansion of $F(z)$. Given two sequences $(a_n)_n$ and $(b_n)_n$ we write $a_n \sim b_n$ to denote the fact that both sequences admit the same asymptotic growth order, specifically $\lim_{n \to \infty} \frac{a_n}{b_n} = 1$. Finally, we write $\varphi \doteq c$ if we are interested in the numerical approximation c of an expression φ.

Suppose that $F(z)$, viewed as a function of a single complex variable z, is defined in some region Ω of the complex plane centred at $z_0 \in \Omega$. Then, if $F(z)$ admits a convergent power series expansion in form of

$$F(z) = \sum_{n \geq 0} f_n (z - z_0)^n \tag{6}$$

it is said to be *analytic* at point z_0. Moreover, if $F(z)$ is analytic at each point $z \in \Omega$, then $F(z)$ is said to be *analytic in the region* Ω. Suppose that there exists a function $G(z)$ analytic in a region Ω^* such that $\Omega \cap \Omega^* \neq \emptyset$ and both $F(z)$ and $G(z)$ agree on $\Omega \cap \Omega^*$, i.e. $F|_{\Omega \cap \Omega^*} = G|_{\Omega \cap \Omega^*}$. Then, $G(z)$ is said to be an *analytic continuation* of $F(z)$ onto Ω^*. If

---

[3] In such a short presentation of a non-trivial theory, many terms, like "branch", "Newton-Puiseux series", "locally convergent" etc. are not defined. They are defined in the references [19, 37, 21].

$F(z)$ defined in some region $\Omega \setminus \{z_0\}$ has no analytic continuation onto Ω, then z_0 is said to be a *singularity* of $F(z)$. When a formal power series $F(z) = \sum_{n \geq 0} f_n z^n$ represents an analytic function in some neighbourhood of the complex plane origin, it becomes possible to link the location and type of singularities corresponding to $F(z)$, in particular so-called *dominating* singularities residing at the respective circle of convergence, with the asymptotic growth rate of its coefficients. This process of *singularity analysis* developed by Flajolet and Odlyzko [18] provides a general and systematic technique for establishing the quantitative aspects of a broad class of combinatorial structures.

While investigating environments and closures, a particular example of algebraic combinatorial structures, the respective generating functions turn out to be algebraic themselves. The following prominent tools provide the essential foundation underlying the process of *algebraic singularity analysis* based on *Newton-Puiseux expansions*, i.e. extensions of power series allowing fractional exponents.

▶ **Theorem 3** (Newton, Puiseux [19, Theorem VII.7]). *Let $F(z)$ be a branch of an algebraic equation $P(z, F(z)) = 0$. Then, in a circular neighbourhood of a singularity ρ slit along a ray emanating from ρ, $F(z)$ admits a fractional Newton-Puiseux series expansion that is locally convergent and of the form*

$$F(z) = \sum_{k \geq k_0} c_k (z - \rho)^{k/\kappa} \tag{7}$$

where $k_0 \in \mathbb{Z}$ and $\kappa \geq 1$.

Let $F(z)$ be analytic at the origin. Note that $[z^n]F(z) = \rho^{-n}[z^n]F(\rho z)$. In consequence, following a proper rescaling we can focus on the type of singularities of $F(z)$ on the unit circle. The standard function scale provides then the asymptotic expansion of $[z^n]F(z)$.

▶ **Theorem 4** (Standard function scale [19, Theorem VI.1]). *Let $\alpha \in \mathbb{C} \setminus \mathbb{Z}_{\leq 0}$. Then, $F(z) = (1 - z)^{-\alpha}$ admits for large n a complete asymptotic expansion in form of*

$$[z^n]F(z) = \frac{n^{\alpha-1}}{\Gamma(\alpha)} \left(1 + \frac{\alpha(\alpha-1)}{2n} + \frac{\alpha(\alpha-1)(\alpha-2)(3\alpha-1)}{24n^2} + O\left(\frac{1}{n^3}\right)\right) \tag{8}$$

where $\Gamma: \mathbb{C} \setminus \mathbb{Z}_{\leq 0} \to \mathbb{C}$ is the Euler Gamma function defined as

$$\Gamma(z) = \int_0^\infty x^{z-1} e^{-x} dx \qquad \text{for } \Re(z) > 0 \tag{9}$$

and by analytic continuation on all its domain.

Given an analytic generating function $F(z)$ implicitly defined as a branch of an algebraic function $P(z, F(z)) = 0$ our task of establishing the asymptotic expansion of the corresponding sequence $([z^n]F(z))_n$ reduces therefore to locating and studying the (dominating) singularities of $F(z)$. For generating functions analytic at the complex plane origin, this quest simplifies even further due to the following classic result.

▶ **Theorem 5** (Pringsheim [19, Theorem IV.6]). *If $F(z)$ is representable at the origin by a series expansion that has non-negative coefficients and radius of convergence R, then the point $z = R$ is a singularity of $F(z)$.*

We can therefore focus on the real line while searching for respective singularities. Since $\sqrt{z}$ cannot be unambiguously defined as an analytic function at $z = 0$ we primarily focus on roots of radicand expressions in the closed-form formulae of investigated generating functions.

4.1 Counting λ-terms

Let us outline the main quantitative results concerning λ-terms in the de Bruijn notation, see [6, 22]. In this combinatorial model, indices are represented in a unary encoding using the successor operator S and 0. In the so-called *natural* size notion [6], assumed throughout the current paper, the size of λ-terms is defined recursively as follows:

$$
\begin{array}{llll}
|0| & = & 1 & \qquad |M\,N| & = & |M|+|N|+1 \\
|\mathsf{S}\,n| & = & |\underline{n}| & = & |n|+1 & \qquad |\lambda M| & = & |M|+1 \,.
\end{array}
$$

And so, for example, $|\lambda\underline{12}| = 7$. We briefly remark that different size notions in the de Bruijn representation, alternative to the assumed natural one, are considered in the literature. Though all share similar asymptotic properties, we choose to consider the above size notion in order to minimise the technical overhead of the overall presentation. We refer the curious reader to [22, 9] for a detailed analysis of various size notions in the de Bruijn representation.

Let l_n denote the number of plain λ-terms of size n. Consider the generating function $L_\infty(z) = \sum_{n \geq 0} l_n z^n$. Using symbolic methods, see [19, Part A. Symbolic Methods] we note that $L_\infty(z)$ satisfies

$$
L_\infty(z) \;=\; zL_\infty(z) + zL_\infty(z)^2 + D(z) \qquad \text{where} \qquad D(z) = \frac{z}{1-z} = \sum_{n=0}^{\infty} z^{n+1}. \tag{10}
$$

In words, a λ-term is either (a) an abstraction followed by another λ-term, accounting for the first summand, (b) an application of two λ-terms, accounting for the second summand, or finally, (c) a de Bruijn index which is, in turn, a sequence of successors applied to 0. Solving (10) for $L_\infty(z)$ we find that the generating function $L_\infty(z)$, taking into account that the coefficients l_n are positive for all n, admits the following closed-form solution:

$$
L_\infty(z) = \frac{1 - z - \sqrt{(1-z)^2 - \frac{4z}{1-z}}}{2z} \,. \tag{11}
$$

In such a form, $L_\infty(z)$ is amenable to the standard techniques of singularity analysis. In consequence we have the following general asymptotic approximation of l_n.

▶ **Theorem 6** (Bendkowski, Grygiel, Lescanne, Zaionc [6]). *The sequence $([z^n]L_\infty(z))_n$ corresponding to plain λ-terms of size n admits the following asymptotic approximation:*

$$
[z^n]L_\infty(z) \sim C\rho_{L_\infty}^{-n} n^{-3/2} \tag{12}
$$

where

$$
\rho_{L_\infty} = \frac{1}{3}\left(\sqrt[3]{26 + 6\sqrt{33}} - \frac{4\,2^{2/3}}{\sqrt[3]{13 + 3\sqrt{33}}} - 1 \right) \doteq 0.29559 \quad and \quad C \doteq 0.60676. \tag{13}
$$

In the context of evaluation, the arguably most interesting subclass of λ-terms are closed or, more generally, m-open λ-terms. Recall that an m-open λ-term takes one of the following forms. Either it is (a) an abstraction followed by an $(m+1)$-open λ-term, or (b) an application of two m-open λ-terms, or finally, (c) one of the indices $\underline{0}, \underline{1}, \ldots, \underline{m-1}$. Such a specification for m-open λ-terms yields the following functional equation defining the associated generating function $L_m(z)$:

$$
L_m(z) = zL_{m+1}(z) + zL_m(z)^2 + \frac{1 - z^m}{1 - z} \,. \tag{14}
$$

Since $L_m(z)$ depends on $L_{m+1}(z)$, solving (14) for $L_m(z)$ one finds that

$$L_m(z) = \frac{1 - \sqrt{1 - 4z^2\left(L_{m+1}(z) + \frac{1-z^m}{1-z}\right)}}{2z} . \tag{15}$$

Such a presentation of $L_m(z)$ poses considerable difficulties as $L_m(z)$ depends on $L_{m+1}(z)$ depending itself on $L_{m+2}(z)$, etc. If developed, the formula (15) for $L_m(z)$ consists of an infinite number of nested radicals. In consequence, standard analytic combinatorics tools do not provide the asymptotic expansion of $[z^n]L_m(z)$, in particular $[z^n]L_0(z)$ associated with closed λ-terms. In their recent breakthrough paper, Bodini, Gittenberger and Gołębiewski [9] propose a clever approximation of the infinite system associated with $L_m(z)$ and give the following asymptotic approximation for the number of m-open λ-terms.

▶ **Theorem 7** (Bodini, Gittenberger and Gołębiewski [9]). *The sequence $([z^n]L_m(z))_n$ corresponding to m-open λ-terms of size n admits the following asymptotic approximation:*

$$[z^n]L_m(z) \sim C_m \rho_{L_\infty}^{-n} n^{-3/2} \tag{16}$$

where ρ_{L_∞} is the dominant singularity corresponding to plain λ-terms, see (13), and C_m is a constant, depending solely on m.

Let us remark that for closed λ-terms, the constant C_0 lies in between 0.07790995266 and 0.0779099823. In what follows, we use the above Theorem 7 in our investigations regarding what we call closed closures.

5 Counting plain closures and environments

In this section we start with counting *plain environments and closures*, i.e. members of $\mathcal{E}nv$ and $\mathcal{C}los$, see (5). We consider a simple model in which the size of environments and closures is equal to the total number of abstractions, applications and the sum of all the de Bruijn index sizes. Formally, we set

$$|\langle M, e\rangle| = |M| + |e| \qquad |\mathfrak{c} : e| = |\mathfrak{c}| + |e| \qquad |\square| = 0 .$$

▶ **Example 8.** The following two tables list the first few plain environments and closures.

size	environments	total
0	$\square$	1
1	$\langle \underline{0}, \square\rangle : \square$	1
2	$\langle \underline{0}, \square\rangle : \langle \underline{0}, \square\rangle : \square$ $\langle \underline{0}, \langle \underline{0}, \square\rangle : \square\rangle : \square$ $\langle \lambda\underline{0}, \square\rangle : \square, \quad \langle \underline{1}, \square\rangle : \square$	4

size	closures	total
0		0
1	$\langle \underline{0}, \square\rangle$	1
2	$\langle \underline{0}, \langle \underline{0}, \square\rangle\rangle$ $\langle \lambda\underline{0}, \square\rangle \quad \langle \underline{1}, \square\rangle$	3

By analogy with the notation $\mathcal{L}_\infty$ for the set of plain λ-terms, we write $\mathcal{E}_\infty$ and $\mathcal{C}_\infty$ to denote the class of plain environments and closures, respectively. Reformulating (5) we can now give a formal specification for both $\mathcal{E}_\infty$ and $\mathcal{C}_\infty$ as follows:

$$\mathcal{E}_\infty = \mathcal{C}_\infty : \mathcal{E}_\infty \ | \ \square$$
$$\mathcal{C}_\infty = \langle \mathcal{L}_\infty, \mathcal{E}_\infty\rangle . \tag{17}$$

In such a form, both classes $\mathcal{E}_\infty$ and $\mathcal{C}_\infty$ become amenable to the process of singularity analysis. In consequence, we obtain the following asymptotic approximation for the number of plain environments and closures.

▶ **Theorem 9.** *The numbers e_n and c_n of plain environments and closures of size n, respectively, admit the following asymptotic approximations:*

$$e_n \sim C_e \cdot \rho^{-n} n^{-3/2} \quad and \quad c_n \sim C_c \cdot \rho^{-n} n^{-3/2} \tag{18}$$

where

$$C_e = \frac{\sqrt{\frac{5}{47}\left(109 + 35\sqrt{545}\right)}}{8\sqrt{\pi}} \doteq 0.699997,$$

$$C_c = \frac{\sqrt{\frac{10\left(48069\sqrt{5} - 10295\sqrt{109}\right)}{65\sqrt{109} - 301\sqrt{5}}}}{\sqrt{\pi}\left(77 - 3\sqrt{545}\right)} \doteq 0.174999 \tag{19}$$

and

$$\rho = \frac{1}{10}\left(25 - \sqrt{545}\right) \doteq 0.165476 \quad giving \quad \rho^{-n} \doteq 6.04315^n. \tag{20}$$

Proof. Consider generating functions $E_\infty(z)$ and $C_\infty(z)$ associated with respective counting sequences, i.e. the sequence $(e_n)_n$ of plain environments of size n and $(c_n)_n$ of plain closures of size n. Based on the specification (17) for $\mathcal{E}_\infty$ and $\mathcal{C}_\infty$ and the assumed size notion, we can write down the following system of functional equations satisfied by $E_\infty(z)$ and $C_\infty(z)$:

$$E_\infty(z) = C_\infty(z)E_\infty(z) + 1$$
$$C_\infty(z) = L_\infty(z)\,E_\infty(z). \tag{21}$$

Next, we solve (21) for $E_\infty(z)$ and $C_\infty(z)$. Though (21) has two formal solutions, the following one is the single one yielding analytic generating functions with non-negative coefficients:

$$E_\infty(z) = \frac{1 - \sqrt{1 - 4L_\infty(z)}}{2L_\infty(z)} \quad and \quad C_\infty(z) = \frac{1}{2}\left(1 - \sqrt{1 - 4L_\infty(z)}\right). \tag{22}$$

Since $L_\infty(z) > 0$ for $z \in (0, \rho_{L_\infty})$ there are two potential sources of singularities in (22). Specifically, the dominating singularity ρ_{L_∞} of $L_\infty(z)$, see (13), or roots of the radicand expression $1 - 4L_\infty(z)$. Therefore, we have to determine whether we fall into the so-called sub- or super-critical composition schema, see [19, Chapter VI. 9]. Solving $1 - 4L_\infty(z) = 0$ for z, we find that it admits a single solution ρ equal to

$$\rho = \frac{1}{10}\left(25 - \sqrt{545}\right) \doteq 0.165476\,. \tag{23}$$

Since $\rho < \rho_{L_\infty}$ the outer radicand carries the dominant singularity ρ of both $E_\infty(z)$ and $C_\infty(z)$. We fall therefore directly into the super-critical composition schema and in consequence know that near ρ both $E_\infty(z)$ and $C_\infty(z)$ admit Newton-Puiseux expansions in form of

$$E_\infty(z) = a_{E_\infty} + b_{E_\infty}\sqrt{1 - \frac{z}{\rho}} + O\left(\left|1 - \frac{z}{\rho}\right|\right)$$

and

$$C_\infty(z) = a_{C_\infty} + b_{C_\infty}\sqrt{1 - \frac{z}{\rho}} + O\left(\left|1 - \frac{z}{\rho}\right|\right) \tag{24}$$

with $a_{E_\infty}, a_{C_\infty} > 0$ and $b_{E_\infty}, b_{C_\infty} < 0$. At this point, we can apply the standard function scale, see Theorem 4, to the presentation of $E_\infty(z)$ and $C_\infty(z)$ in (24) and conclude that

$$[z^n]E_\infty(z) \sim C_{E_\infty}\rho^{-n}n^{-3/2} \quad \text{and} \quad [z^n]C_\infty(z) \sim C_{C_\infty}\rho^{-n}n^{-3/2} \tag{25}$$

where $C_{E_\infty} = \dfrac{b_{E_\infty}}{\Gamma(-\frac{1}{2})}$ and $C_{C_\infty} = \dfrac{b_{C_\infty}}{\Gamma(-\frac{1}{2})}$, respectively, with $\Gamma(-\frac{1}{2}) = 2\sqrt{\pi}$. In fact, reformulating (22) so to fit the Newton-Puiseux expansion forms (24) we find that

$$a_{E_\infty} = 2, \quad b_{E_\infty} = -\frac{1}{4}\sqrt{\frac{5}{47}\left(109 + 35\sqrt{545}\right)} \tag{26}$$

and

$$a_{C_\infty} = \frac{1}{2}, \quad b_{C_\infty} = \frac{2\sqrt{\frac{10\left(48069\sqrt{5}-10295\sqrt{109}\right)}{65\sqrt{109}-301\sqrt{5}}}}{3\sqrt{545} - 77} \tag{27}$$

Numerical approximations of $C_{E_\infty} = \dfrac{b_{E_\infty}}{\Gamma(-\frac{1}{2})}$ and $C_{C_\infty} = \dfrac{b_{C_\infty}}{\Gamma(-\frac{1}{2})}$ yield the declared asymptotic behaviour of $(e_n)_n$ and $(c_n)_n$, see (18). ◀

Let us notice that as both generating functions $E_\infty(z)$ and $C_\infty(z)$ are algebraic, they are also *holonomic* (D-finite), i.e. satisfy differential equations with polynomial (in terms of z) coefficients. Using the powerful `gfun` library for `Maple` [34] one can automatically derive appropriate holonomic equations for $E_\infty(z)$ and $C_\infty(z)$, subsequently converting them into linear recurrences for sequences $(e_n)_n$ and $(c_n)_n$.

▶ **Example 10.** We restrict the presentation to the linear recurrence for the number of plain environments, omitting for brevity the, likely verbose, respective recurrence for plain closures. Using `gfun` we find that e_n satisfies the recurrence of Figure 2. Despite its appearance, this recurrence is an efficient way of computing e_n. Indeed, holonomic specifications for $C_\infty(z)$ and $E_\infty(z)$ allow computing the coefficients $[z^n]C_\infty(z)$ and $[z^n]E_\infty(z)$ using a linear number of arithmetic operations, as opposed to a quadratic number of operations as following their direct combinatorial specification. Let us remark that the computations involved operate on large, having linear in n space representation, integers. For instance, e_{1000} has about 600 digits. In consequence, single arithmetic operations on such numbers cannot be performed in constant time.

▶ Remark. The analytic approach utilising generating functions exhibits an important benefit in the context of generating random instances of plain environments and closures. With analytic generating functions at hand and effective means of computing both $[z^n]E_\infty(z)$ and $[z^n]C_\infty(z)$, it is possible to design efficient samplers, constructing uniformly random (conditioned on the outcome size n) structures of both combinatorial classes. For instance, using holonomic specifications it becomes possible to construct exact-size samplers following the so-called recursive method of Nijenhuis and Wilf, see [31, 20]. Moreover, since corresponding generating functions are analytic, it is possible to design effective Boltzmann samplers [17], either in their approximate-size variant constructing structures within a structure size interval $[(1-\varepsilon)n, (1+\varepsilon)n]$ in time $O(|\omega|)$ where ω is the outcome structure, or their exact-size variants running in time $O(n^2)$. Remarkably, both sampler frameworks admit effective tuning procedures influencing the expected internal shape of constructed objects, e.g. frequencies of desired sub-patterns [5]. With the growing popularity of (semi-)automated software testing

$$(125\,n^3 - 125\,n)\,e_n +$$
$$(-475\,n^3 - 150\,n^2 + 325\,n)\,e_{n+1} +$$
$$(-1625\,n^3 - 13650\,n^2 - 29125\,n - 17100)\,e_{n+2} +$$
$$(5925\,n^3 + 65550\,n^2 + 204825\,n + 190800)\,e_{n+3} +$$
$$(-10950\,n^3 - 149850\,n^2 - 609000\,n - 744300)\,e_{n+4} +$$
$$(43599\,n^3 + 638460\,n^2 + 3028701\,n + 4633680)\,e_{n+5} +$$
$$(-97781\,n^3 - 1680378\,n^2 - 9481237\,n - 17550960)\,e_{n+6} +$$
$$(122749\,n^3 + 2388066\,n^2 + 15211685\,n + 31648968)\,e_{n+7} +$$
$$(-184402\,n^3 - 3954630\,n^2 - 27717140\,n - 63149544)\,e_{n+8} +$$
$$(280081\,n^3 + 6826380\,n^2 + 54868451\,n + 145130568)\,e_{n+9} +$$
$$(-205649\,n^3 - 5654610\,n^2 - 51851989\,n - 158722620)\,e_{n+10} +$$
$$(37439\,n^3 + 1339686\,n^2 + 16635271\,n + 70682784)\,e_{n+11} +$$
$$(-68686\,n^3 - 3028038\,n^2 - 43616336\,n - 205972920)\,e_{n+12} +$$
$$(222029\,n^3 + 9258780\,n^2 + 128417911\,n + 592399800)\,e_{n+13} +$$
$$(-241115\,n^3 - 10519830\,n^2 - 152823475\,n - 739190880)\,e_{n+14} +$$
$$(134151\,n^3 + 6201222\,n^2 + 95476551\,n + 489605640)\,e_{n+15} +$$
$$(-42231\,n^3 - 2067834\,n^2 - 33729375\,n - 183277332)\,e_{n+16} +$$
$$(7470\,n^3 + 386418\,n^2 + 6659316\,n + 38233296)\,e_{n+17} +$$
$$(-678\,n^3 - 36972\,n^2 - 671670\,n - 4065240)\,e_{n+18} +$$
$$(24\,n^3 + 1380\,n^2 + 26436\,n + 168720)\,e_{n+19} = 0.$$

$$
\begin{aligned}
e_0 &= 1, & e_{10} &= 1233816, \\
e_1 &= 1, & e_{11} &= 6558106, \\
e_2 &= 4, & e_{12} &= 35202448, \\
e_3 &= 17, & e_{13} &= 190568779, \\
e_4 &= 77, & e_{14} &= 1039296373, \\
e_5 &= 364, & e_{15} &= 5704834700, \\
e_6 &= 1776, & e_{16} &= 31494550253, \\
e_7 &= 8881, & e_{17} &= 174759749005, \\
e_8 &= 45296, & e_{18} &= 974155147162. \\
e_9 &= 234806, &
\end{aligned}
$$

Figure 2 Linear recurrence defining e_n with corresponding initial conditions.

techniques, see [12], combinatorial samplers for environments and closures exhibit potential applications in testing normalisation frameworks and abstract machines implementations, such as the Krivine machine. We briefly remark that randomly generated λ-terms already proved useful in finding optimisation bugs in compilers of functional programming languages, see [32]. Our prototype samplers for environments and closures, within above sampler frameworks, are available at Github[4].

6   Counting closed closures

In this section we address the problem of counting so-called *closed closures*[5]. A closure is said to be *closed* if it consists of an m-open term associated with an environment of length m made itself out of closed closures. Note that such closures correspond to not yet fully evaluated m-open λ-terms. With such a description, the set $\mathcal{C}los_0$ of closed closures can be given using the following combinatorial specification:

$$\mathcal{C}los_0 ::= \mathcal{L}_0 \times \square \mid \mathcal{L}_1 \times \langle \mathcal{C}los_0 \rangle \mid \mathcal{L}_2 \times \langle \mathcal{C}los_0, \mathcal{C}los_0 \rangle \mid \mathcal{L}_3 \times \langle \mathcal{C}los_0, \mathcal{C}los_0, \mathcal{C}los_0 \rangle \mid \cdots \quad (28)$$

---

[4] `https://github.com/PierreLescanne/CountingAndGeneratingClosuresAndEnvironments`
[5] We acknowledge that speaking of closed closures is a bit odd, however terms "closure" and "closed" form a consecrated terminology that we merely associate together.

▶ **Example 11.** The following table lists the first few closed closures.

size	closures	total
0, 1		0
2	$\langle \lambda \underline{0}, \square \rangle$	1
3	$\langle \lambda \lambda \underline{0}, \square \rangle$ $\langle \underline{0}, \langle \lambda \underline{0}, \square \rangle \rangle$	2
4	$\langle \lambda \lambda \lambda \underline{0}, \square \rangle$ $\langle \lambda \lambda \underline{1}, \square \rangle$ $\langle \lambda (\underline{0}\underline{0}), \square \rangle$ $\langle \lambda \underline{0}, \langle \lambda \underline{0}, \square \rangle \rangle$ $\langle \underline{0}, \langle \lambda \lambda \underline{0}, \square \rangle \rangle$ $\langle \underline{0}, \langle \underline{0}, \langle \lambda \underline{0}, \square \rangle \rangle \rangle$	6

Establishing the asymptotic growth rate of the sequence $(c_{0,n})_n$ corresponding to closed closures of size n poses a considerable challenge, much more involved than its plain counterpart. In the following theorem we show that there exists two constants $\underline{\rho}, \overline{\rho} < \rho_{L_\infty}$ such that $\lim_{n \to \infty} \frac{\underline{\rho}^{-n}}{c_{0,n}} = 0$ and $\lim_{n \to \infty} \frac{c_{0,n}}{\overline{\rho}^{-n}} = 0$. In other words, the asymptotic growth rate of $(c_{0,n})_n$ is bounded by two exponential functions of n.

▶ **Theorem 12.** *There exist $\overline{\rho} < \underline{\rho}$ satisfying $\overline{\rho} < \underline{\rho} < \rho_{L_\infty}$ and functions $\theta(n), \kappa(n)$ satisfying $\limsup_{n \to \infty} \theta(n)^{1/n} = \limsup_{n \to \infty} \kappa(n)^{1/n} = 1$ such that for sufficiently large n we have $\underline{\rho}^{-n} \theta(n) < c_{0,n} < \overline{\rho}^{-n} \kappa(n)$.*

Proof. Let us start with the generating function $C_0(z)$ associated with closed closures $\mathcal{C}los_0$. Note that from the specification (28) $C_0(z)$ is implicitly defined as

$$C_0(z) = \sum_{m \geq 0} L_m(z) C_0(z)^m. \tag{29}$$

We can therefore identify a closed closure $\mathfrak{c}$ with a tuple $(t, c_1, \ldots, c_m)$ where $m \geq 0$, t is an m-open λ-term and $c_1, \ldots, c_m$ are closed closures themselves. We proceed with defining two auxiliary lower and upper bound classes $\underline{C}_0(z)$ and $\overline{C}_0(z)$ such that $[z^n]\underline{C}_0(z) \leq [z^n]C_0(z) \leq \overline{C}_0(z)$ for all n. Next, we establish their asymptotic behaviour and, in doing so, provide exponential lower and upper bounds on the growth rate of closed closures.

We start with $\underline{C}_0(z) = \sum_{m \geq 0} L_0(z) \underline{C}_0(z)^m$. Note that $\underline{C}_0(z)$ is associated with closures in which each term is closed, independently of the corresponding environment length. Hence, as closed λ-terms are m-open for all $m \geq 0$, we have $[z^n]\underline{C}_0(z) \leq [z^n]C_0(z)$. Furthermore

$$\underline{C}_0(z) = \sum_{m \geq 0} L_0(z) \underline{C}_0(z)^m = L_0(z) \sum_{m \geq 0} \underline{C}_0(z)^m = \frac{L_0(z)}{1 - \underline{C}_0(z)}. \tag{30}$$

Solving the above equation for $\underline{C}_0(z)$ we find that $\underline{C}_0(z) = \frac{1}{2}\left(1 - \sqrt{1 - 4L_0(z)}\right)$. In such a form, it is clear that there are two potential sources of singularities, i.e. the singularity ρ_{L_∞} of $L_0(z)$, see Theorem 7, or the roots of the radicand $1 - 4L_0(z)$. Since $L_0(z)$ is increasing and continuous in the interval $(0, \rho_{L_\infty})$ we know that if $L_0(\rho_{L_\infty}) > \frac{1}{4}$ then there exists a $\underline{\rho} < \rho_{L_\infty}$ such that $L_0(\underline{\rho}) = \frac{1}{4}$. Unfortunately, we cannot simply check that $L_0(\underline{\rho}) > \frac{1}{4}$ as there exists no known method of evaluating $L_0(z)$, defined by means of an infinite system of equations, at a given point. For that reason we propose the following approach.

Recall that a λ-term M is said to be *h-shallow* if all its de Bruijn index values are (strictly) bounded by h, see [22]. Let $L_m^{(h)}(z)$ denote the generating function associated with m-open h-shallow λ-terms. Note that $L_0^{(h)}(z)$, i.e. the generating function corresponding to closed

h-shallow λ-terms, has a finite computable representation. Indeed, we have

$$L_0^{(h)}(z) = zL_1^{(h)}(z) + zL_0^{(h)}(z)L_0^{(h)}(z)$$
$$L_1^{(h)}(z) = zL_2^{(h)}(z) + zL_1^{(h)}(z)L_1^{(h)}(z) + z$$
$$L_2^{(h)}(z) = zL_3^{(h)}(z) + zL_2^{(h)}(z)L_2^{(h)}(z) + z + z^2$$
$$\cdots$$
$$L_{h-1}^{(h)}(z) = zL_h^{(h)}(z) + zL_{h-1}^{(h)}(z)L_{h-1}^{(h)}(z) + z + z^2 + \cdots + z^{h-1}$$
$$L_h^{(h)}(z) = zL_h^{(h)}(z) + zL_h^{(h)}(z)L_h^{(h)}(z) + z + z^2 + \cdots + z^h \tag{31}$$

Consider $m < h$. Each m-open h-shallow λ-term is either (a) in form of λM where M is an $(m+1)$-open h-shallow λ-term due to the head abstraction, (b) in form of MN where both M and N are m-open h-shallow λ-terms, or (c) a de Bruijn index in the set $\{\underline{0}, \underline{1}, \ldots, \underline{m-1}\}$. When $m = h$, we have the same specification with the exception of the first summand $zL_h^{(h)}(z)$ where, as we cannot exceed h, terms under abstractions are h-open, instead of $(h+1)$-open.

Using such a form it is possible to evaluate $L_0^{(h)}(z)$ at each point $z \in (0, \rho_{(h)})$ where $\rho_{(h)} > \rho_{L_\infty}$ is the dominating singularity of $L_0^{(h)}(z)$ satisfying $\rho_{(h)} \xrightarrow{h \to \infty} \rho$, see [22]. Certainly, each closed h-shallow λ-term is in particular a closed λ-term. In consequence, $[z^n]L_0^{(h)}(z) \leq [z^n]L_0(z)$ for each n. Moreover, for all sufficiently large n we have $[z^n]L_0^{(h)}(z) < [z^n]L_0(z)$. This coefficient-wise lower bound transfers onto the level of generating function values and we obtain $L_0^{(h)}(z) < L_0(z)$. Following the same argument, we also have $L_0^{(h)}(z) < L_0^{(h+1)}(z)$ for each $h \geq 1$. We can therefore use $L_0^{(h)}(z)$ to approximate $L_0(z)$ from below — the higher h we choose, the better approximation we obtain. Using computer algebra software[6] it is possible to automatise the evaluation process of $L_0^{(h)}(\rho_{L_\infty})$ for increasing values of h and find that for $h = 153$ we obtain

$$L_0^{(153)}(\rho_{L_\infty}) \doteq 0.25000324068941554. \tag{32}$$

Hence indeed, the asserted existence of $\rho < \rho_{L_\infty}$ such that $L_0(\rho) = \frac{1}{4}$ follows (interestingly, taking $h = 152$ does not suffice as $L_0^{152}(\rho_{L_\infty}) < \frac{1}{4}$). We fall hence in the super-critical composition schema[7] and note that $\underline{C}_0(z)$ admits a Newton-Puiseux expansion near $\underline{\rho}$ as follows:

$$\underline{C}_0(z) = \underline{a_0} - \underline{b_0}\sqrt{1 - \frac{z}{\rho}} + O\left(\left|1 - \frac{z}{\rho}\right|\right) \tag{33}$$

for some constants $\underline{a_0} > 0$ and $\underline{b_0} < 0$. Hence, $[z^n]C_0(z)$ grows asymptotically faster than $\underline{\rho}^{-n}\theta(n)$ where $\theta(n) = \frac{b_0}{\Gamma(-\frac{1}{2})}n^{-3/2}$.

For the upper bound we consider $\overline{C}_0(z) = \sum_{m \geq 0} L_\infty(z)\overline{C}_0(z)^m$, i.e. the generating function associated with closures in which all terms are plain (either closed or open), independently of the constraint imposed by the corresponding environment length. Following the same arguments as before, we note that $[z^n]\overline{C}_0(z) > [z^n]C_0(z)$. Now

$$\overline{C}_0(z) = \sum_{m \geq 0} L_\infty(z)\overline{C}_0(z)^m = L_\infty(z)\sum_{m \geq 0}\overline{C}_0(z)^m = \frac{L_\infty(z)}{1 - \overline{C}_0(z)}. \tag{34}$$

---

[6] `https://github.com/PierreLescanne/CountingAndGeneratingClosuresAndEnvironments`
[7] *Supercriticality* ensures that meromorphic asymptotics applies and entails strong statistical regularities (see [19] Section V.2 and Section IX.6).

Solving the equation for $\overline{C}_0(z)$ we find that $\overline{C}_0(z) = \frac{1}{2}\left(1 - \sqrt{1 - 4L_\infty(z)}\right)$. Note that in this case, we can easily handle the radicand expression $1 - 4L_\infty(z)$ and find out that, as in the lower bound case, we are in the super-critical composition schema. Specifically, $\overline{\rho} = \frac{1}{10}\left(25 - \sqrt{545}\right) \doteq 0.165476$, cf. (20), is the dominating singularity of $\overline{C}_0(z)$. In consequence, $\overline{C}_0(z)$ admits the following Newton-Puiseux expansion near $\overline{\rho}$:

$$\overline{C}_0(z) = \overline{a_0} - \overline{b_0}\sqrt{1 - \frac{z}{\overline{\rho}}} + O\left(\left|1 - \frac{z}{\overline{\rho}}\right|\right) \tag{35}$$

for some constants $\overline{a_0} > 0$ and $\overline{b_0} < 0$. In conclusion, $[z^n]C_0(z)$ grows asymptotically slower than $(\overline{\rho})^{-n}\theta(n)$ where $\theta(n) = \dfrac{\overline{b_0}}{\Gamma(-\frac{1}{2})}n^{-3/2}$, finishing the proof. ◄

With an implicit expression defining $C_0(z)$, see (29), efficient random generation of closed closures poses a difficult task. Though we have no efficient Boltzmann samplers, it is possible to follow the recursive method and an obtain exact-size samplers for a moderate range of target sizes. We offer a prototype sampler of this kind, available at Github[8].

7　Conclusions

We view our contribution as a small step towards the quantitative, average-case analysis of evaluation complexity in λ-calculus. Using standard tools from analytic combinatorics, we investigated some combinatorial aspects of environments and closures — fundamental structures present in various formalisms dealing with normalisation in λ-calculus, especially in its variants with explicit substitutions [28]. Though plain environments and closures are relatively easy to count and generate, their closed counterparts pose a considerable combinatorial challenge. The implicit and infinite specification of closed closures based on closed λ-terms complicates significantly the quantitative analysis, namely estimating the exponential factor in the asymptotic growth rate, or effectively generating random closed closures. In particular, getting more parameters of the asymptotic growth will require more sophisticated methods, like, for instance, the recent infinite system approximation techniques of Bodini, Gittenberger and Gołębiewski [9].

───── **References** ─────

1 Beniamino Accattoli and Ugo Dal Lago. (leftmost-outermost) beta reduction is invariant, indeed. *Logical Methods in Computer Science*, 12(1), 2016. `doi:10.2168/LMCS-12(1:4)2016`.

2 Martin Avanzini, Ugo Dal Lago, and Georg Moser. Analysing the complexity of functional programs: higher-order meets first-order. In Kathleen Fisher and John H. Reppy, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada.*, pages 152–164. ACM, 2015. `doi:10.1145/2784731.2784753`.

3 Martin Avanzini and Georg Moser. Closing the gap between runtime complexity and poly-time computability. In Christopher Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA 2010, July 11-13, 2010, Edinburgh, Scottland, UK*, volume 6 of *LIPIcs*, pages 33–48. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010. `doi:10.4230/LIPIcs.RTA.2010.33`.

---

[8] `https://github.com/PierreLescanne/CountingAndGeneratingClosuresAndEnvironments`

4 Maciej Bendkowski. Normal-order reduction grammars. *Journal of Functional Programming*, 27, 2017. `doi:10.1017/S0956796816000332`.

5 Maciej Bendkowski, Olivier Bodini, and Sergey Dovgal. *Polynomial tuning of multiparametric combinatorial samplers*, pages 92–106. SIAM, 2018. `doi:10.1137/1.9781611975062.9`.

6 Maciej Bendkowski, Katarzyna Grygiel, Pierre Lescanne, and Marek Zaionc. Combinatorics of λ-terms: a natural approach. *Journal of Logic and Computation*, 27(8):2611–2630, 2017. `doi:10.1093/logcom/exx018`.

7 Maciej Bendkowski, Katarzyna Grygiel, and Marek Zaionc. On the likelihood of normalization in combinatory logic. *Journal of Logic and Computation*, 2017. `doi:10.1093/logcom/exx005`.

8 Olivier Bodini, Danièle Gardy, and Bernhard Gittenberger. Lambda-terms of bounded unary height. In Philippe Flajolet and Daniel Panario, editors, *Proceedings of the Eighth Workshop on Analytic Algorithmics and Combinatorics, ANALCO 2011, San Francisco, California, USA, January 22, 2011*, pages 23–32. SIAM, 2011. `doi:10.1137/1.9781611973013.3`.

9 Olivier Bodini, Bernhard Gittenberger, and Zbigniew Gołębiewski. Enumerating lambda terms by weighted length of their de bruijn representation. *CoRR*, abs/1707.02101, 2017. URL: `https://arxiv.org/abs/1707.02101`.

10 Christine Choppy, Stéphane Kaplan, and Michèle Soria. Algorithmic complexity of term rewriting systems. In Pierre Lescanne, editor, *Rewriting Techniques and Applications, 2nd International Conference, RTA-87, Bordeaux, France, May 25-27, 1987, Proceedings*, volume 256 of *Lecture Notes in Computer Science*, pages 256–273. Springer, 1987. `doi:10.1007/3-540-17220-3_22`.

11 Christine Choppy, Stéphane Kaplan, and Michèle Soria. Complexity analysis of term-rewriting systems. *Theor. Comput. Sci.*, 67(2&3):261–282, 1989. `doi:10.1016/0304-3975(89)90005-4`.

12 Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 268–279. ACM, 2000.

13 Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms, and Functional Programming (2nd Ed.)*. Birkhauser Boston Inc., Cambridge, MA, USA, 1994.

14 Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):362–397, March 1996. `doi:10.1145/226643.226675`.

15 René David, Katarzyna Grygiel, Jakub Kozik, Christophe Raffalli, Guillaume Theyssier, and Marek Zaionc. Asymptotically almost all λ-terms are strongly normalizing. *Logical Methods in Computer Science*, 9:1–30, 2013.

16 Nicolaas G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972.

17 Philippe Duchon, Philippe Flajolet, Guy Louchard, and Gilles Schaeffer. Boltzmann samplers for the random generation of combinatorial structures. *Combinatorics, Probability and Computing*, 13(4-5):577–625, 2004.

18 Philippe Flajolet and Andrew M. Odlyzko. Singularity analysis of generating functions. *SIAM Journal on Discrete Mathematics*, 3(2):216–240, 1990.

19 Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 1 edition, 2009.

20 Philippe Flajolet, Paul Zimmermann, and Bernard Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theoretical Computer Science*, 132(1):1–35, 1994.

21 Étienne Ghys. *A singular mathematical promenade*. Ecole Normale Supérieure, 2017. URL: `http://perso.ens-lyon.fr/ghys/promenade/`.

22 Bernhard Gittenberger and Zbigniew Gołębiewski. On the number of lambda terms with prescribed size of their de Bruijn representation. In *33rd Symposium on Theoretical Aspects of Computer Science, STACS*, pages 40:1–40:13, 2016.

23 Katarzyna Grygiel and Pierre Lescanne. Counting and generating terms in the binary lambda calculus. *Journal of Functional Programming*, 25, 2015. `doi:10.1017/S0956796815000271`.

24 Donald E. Knuth. Mathematical Analysis of Algorithms, 2000. First chapter of [25].

25 Donald E. Knuth. *Selected Papers on Analysis of Algorithms*, volume 102 of *CSLI Lecture Notes*. Stanford, California: Center for the Study of Language and Information, 2000.

26 Ugo Dal Lago and Simone Martini. On constructor rewrite systems and the lambda calculus. *Logical Methods in Computer Science*, 8(3), 2012. `doi:10.2168/LMCS-8(3:12)2012`.

27 Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964. `doi:10.1093/comjnl/6.4.308`.

28 Pierre Lescanne. From $\lambda\sigma$ to $\lambda\upsilon$: A journey through calculi of explicit substitutions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 60–69. ACM, 1994.

29 Michel Mauny and Ascánder Suárez. Implementing functional languages in the categorical abstract machine. In *LISP and Functional Programming*, pages 266–278, 1986.

30 John C. Mitchell. *Concepts in Programming Language (1st Ed.)*. Cambridge University Press, New York, NY, USA, 2002.

31 Albert Nijenhuis and Herbert S. Wilf. *Combinatorial Algorithms*. Academic Press, 2 edition, 1978.

32 Michał H. Pałka. *Random Structured Test Data Generation for Black-Box Testing*. PhD thesis, Chalmers University of Technology, 2012.

33 Gordon David Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975. `doi:10.1016/0304-3975(75)90017-1`.

34 Bruno Salvy and Paul Zimmermann. Gfun: a Maple package for the manipulation of generating and holonomic functions in one variable. *ACM Transactions on Mathematical Software*, 20(2):163–177, 1994.

35 Robert Sedgewick and Philippe Flajolet. *An Introduction to the Analysis of Algorithms (2nd Edition)*. Createspace Independent Pub, 2014.

36 Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *J. ACM*, 51(3):385–463, 2004. `doi:10.1145/990308.990310`.

37 Herbert S. Wilf. *Generatingfunctionology*. A. K. Peters, Ltd., 2006.

Higher-Order Equational Pattern Anti-Unification

David M. Cerna
Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria
david.cerna@risc.jku.at

Temur Kutsia
Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria
temur.kutsia@risc.jku.at

—— **Abstract** ————————————————————————————

We consider anti-unification for simply typed lambda terms in associative, commutative, and associative-commutative theories and develop a sound and complete algorithm which takes two lambda terms and computes their generalizations in the form of higher-order patterns. The problem is finitary: the minimal complete set of generalizations contains finitely many elements. We define the notion of optimal solution and investigate special fragments of the problem for which the optimal solution can be computed in linear or polynomial time.

1 Introduction

Anti-unification algorithms aim at computing generalizations for given terms. A generalization of t and s is a term r such that t and s are substitution instances of r. Interesting generalizations are those that are least general (lggs). However, it is not always possible to have a unique least general generalization. In these cases the task is either to compute a minimal complete set of generalizations, or to impose restrictions so that uniqueness is guaranteed.

Anti-unification, as considered in this paper, uses both of these ideas. The theory is simply-typed lambda calculus, where some function symbols may be associative, commutative, or associative-commutative. A-, C-, and AC-anti-unification is finitary even for first-order terms, and a modular algorithm has been proposed in [1] to compute the corresponding minimal complete set of generalizations. Anti-unification for simply typed lambda terms can be restricted to compute generalizations in the form of Miller's patterns [13], which makes it unitary, and the single least general generalization can be computed in linear time by the algorithm proposed in [8]. These two approaches combine nicely with each other when one wants to develop a higher-order equational anti-unification algorithm, and we illustrate it in this paper. Basically, it extends the syntactic[1] generalization rules from [8] by equational decomposition rules inspired by those from [1], getting a modular algorithm in which different equational axioms for different function symbols can be combined automatically. The

---

[1] We refer to the higher-order anti-unification algorithm from [8] as syntactic, although it works modulo $\beta\eta$-conversion.

algorithm takes a pair of simply typed lambda terms and returns a set of their generalizations in the form of higher-order patterns. It is terminating, sound, and complete. However, the number of nondeterministic choices when decomposing may result in a large search tree. Although each branch can be developed in linear time, there can be too many of them to search efficiently.

This is the problem that we address in the second part of the paper. The idea is to use a greedy approach: introduce an optimality criterion, use it to select an anti-unification problem among different alternatives obtained by a decomposition rule, and try to solve only that. In this way, we would only compute one generalization. Checking the criterion and selecting the right branch should be done "reasonably fast". To implement this idea, we introduce conditions on the form of anti-unification problems which are guarantee to compute "optimal" solutions, and study the corresponding complexities. In particular, we identify conditions for which A-, C-, and AC-generalizations can be computed in linear time. We also study how the complexity changes by relaxing these conditions.

Higher-order anti-unification has been investigated by various authors from different application perspective. Research has been focused mainly on the investigation of special classes for which the uniqueness of lgg is guaranteed. Some application areas include proof generalization [14], higher-order term indexing [15], cognitive modeling and analogical reasoning [9, 17], recursion scheme detection in functional programs [3], inductive synthesis of recursive functions [16], just to name a few. Two higher-order anti-unification algorithms [6, 8] are included in an online open-source anti- unification library [4, 5]. This related work does not consider anti-unification with higher-order terms in the presence of equational axioms. However, such a combination can be useful, for instance, for developing indexing techniques for higher-order theorem provers [12] or in higher order program manipulation tools.

The organization of the paper is as follows: In Section 2 we introduce the main notions and define the problem. In Section 3 we recall the higher-order anti-unification algorithm from [8]. In Section 4 we extend the algorithm with equational decomposition rules. Section 5 is devoted to the introduction of computationally well-behaved fragments of anti-unification problems. The next sections describe the behavior of equational anti-unification algorithms on these fragments: In Section 6 we discuss associative generalization and speak about optimality. Sections 7 and 8 are about C- and AC-generalizations. Sections 9 summarizes the results and contains a discussion of future work and open problems.

2   Preliminaries

This work builds upon the formulations and results of [7, 8]. Higher-order signatures are composed of *types* constructed from a set of *base types* (typically δ) using the grammar $\tau ::= \delta \mid \tau \to \tau$. We will consider $\to$ to be associative right unless otherwise stated. *Variables* (typically $X, Y, Z, x, y, z, a, b, \ldots$) as well as *constants* (typically $f, c, \ldots$) are assigned types from the set of types constructed using the above grammar. λ-*terms* (typically $t, s, u, \ldots$) are constructed using the grammar $t ::= x \mid c \mid \lambda x.t \mid t_1\ t_2$ where x is a variable and c is a constant, and are typed using the type construction mentioned above. Terms of the form $(\ldots (h\ t_1) \ldots t_m)$, where h is a constant or a variable, will be written as $h(t_1, \ldots, t_m)$, and terms of the form $\lambda x_1. \ldots . \lambda x_n.t$ as $\lambda x_1, \ldots, x_n.t$. We use $\vec{x}$ as a short-hand for $x_1, \ldots, x_n$. This basic language will be extended by higher-order constants satisfying equational axioms. When necessary, we write a λ-term t together with its type α as $t : \alpha$.

Every higher-order constant c will have an associated set of axioms, denoted by $Ax(c)$. If $Ax(c)$ is empty then c does not have any associated properties and is called *free*. Otherwise,

$Ax(f) \subseteq \{A, C\}$ where A is associativity, i.e. $f(a, f(b, c)) \equiv f(f(a, b), c)$ and C is commutativity, i.e. $f(a, b) \equiv f(b, a)$. Note that only functions of the type $\alpha \to \alpha \to \alpha$ are allowed to have equational properties. We assume that terms are written in *flattened form*, obtained by replacing all subterms of the form $f(t_1, \ldots, f(s_1, \ldots, s_m), \ldots t_n)$ by $f(t_1, \ldots, s_1, \ldots, s_m, \ldots t_n)$, where $A \in Ax(f)$. Also, by convention, the term $f(t)$ stands for t, if $A \in Ax(f)$. Other standard notions of the simply typed λ-calculus, like bound and free occurrences of variables, α-conversion, β-reduction, η-long β-normal form, etc. are defined as usual (see [2, 10]). By default, terms are assumed to be written in η-long β-normal form. Therefore, all terms have the form $\lambda x_1, \ldots, x_n.h(t_1, \ldots, t_m)$, where $n, m \geq 0$, h is either a constant or a variable, $t_1, \ldots, t_m$ have this form, and the term $h(t_1, \ldots, t_m)$ has a basic type.

The set of free variables of a term t is denoted by $\mathrm{Vars}(t)$. When we write an equality between two λ-terms, we mean that they are equivalent modulo α, β and η equivalence.

The *size* of a term t, denoted $|t|$, is defined recursively as $|h(t_1, \ldots, t_n)| = 1 + \sum_{i=1}^{n} |t_i|$ and $|\lambda x.t| = 1 + |t|$. The *depth* of a term t, denoted $depth(t)$ is defined recursively as $depth(h(t_1, \ldots, t_n)) = 1 + \max_{i \in \{1, \ldots, n\}} depth(t_i)$ and $depth(\lambda x.t) = 1 + depth(t)$. For a term $t = \lambda x_1, \ldots, x_n.h(t_1, \ldots, t_m)$ with $n, m \geq 0$, its *head* is defined as $head(t) = h$.

A *higher-order pattern* is a λ-term where, when written in η-long β-normal form, all free variable occurrences are applied to lists of pairwise distinct (η-long forms of) bound variables. For instance, $\lambda x.f(X(x), Y)$, $f(c, \lambda x.x)$ and $\lambda x.\lambda y.X(\lambda z.x(z), y)$ are patterns, while $\lambda x.f(X(X(x)), Y)$, $f(X(c), c)$ and $\lambda x.\lambda y.X(x, x)$ are not.

Substitutions are finite sets of pairs $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ where X_i and t_i have the same type and the X's are pairwise distinct variables. They can be extended to type preserving functions from terms to terms as usual, avoiding variable capture. The notions of substitution *domain* and *range* are also standard and are denoted, respectively, by Dom and Ran.

We use postfix notation for substitution applications, writing $t\sigma$ instead of $\sigma(t)$. As usual, the application $t\sigma$ affects only the free occurrences of variables from $\mathrm{Dom}(\sigma)$ in t. We write $\vec{x}\sigma$ for $x_1\sigma, \ldots, x_n\sigma$, if $\vec{x} = x_1, \ldots, x_n$. Similarly, for a set of terms S, we define $S\sigma = \{t\sigma \mid t \in S\}$. The *composition* of σ and ϑ is written as juxtaposition $\sigma\vartheta$ and is defined as $x(\sigma\vartheta) = (x\sigma)\vartheta$ for all x. Another standard operation, *restriction* of a substitution σ to a set of variables S, is denoted by $\sigma|_S$.

A substitution σ_1 is *more general* than σ_2, written $\sigma_1 \preceq \sigma_2$, if there exists ϑ such that $X\sigma_1\vartheta = X\sigma_2$ for all $X \in \mathrm{Dom}(\sigma_1) \cup \mathrm{Dom}(\sigma_2)$. The strict part of this relation is denoted by $\prec$. The relation $\preceq$ is a partial order and generates the equivalence relation which we denote by $\simeq$. We overload $\preceq$ by defining $s \preceq t$ if there exists a substitution σ such that $s\sigma = t$. The focus of this work is generalization in the presence of equational axioms thus we need a more general concept of ordering of substitutions/terms by their generality. We say that two terms s, t are $s =_{\mathcal{E}} t$ if they are equivalent modulo $\mathcal{E} \subseteq \{A, C\}$. For example, $f(a, f(b, c)) \neq f(f(a, b), c)$ but, $f(a, f(b, c)) =_{\{A\}} f(f(a, b), c)$. Under this notion of equality we can say that a substitution σ_1 is *more general modulo an equational theory $\mathcal{E} \subseteq \{A, C\}$* than σ_2 written $\sigma_1 \preceq_{\mathcal{E}} \sigma_2$ if there exists ϑ such that $X\sigma_1\vartheta =_{\mathcal{E}} X\sigma_2$ for all $X \in \mathrm{Dom}(\sigma_1) \cup \mathrm{Dom}(\sigma_2)$ Note that $\prec$ and $\simeq$ and the term extension are generalized accordingly. Form this point on we will use the ordering relation modulo an equational theory when discussing generalization.

A term t is called a *generalization* or an *anti-instance* modulo an equational theory $\mathcal{E}$ of two terms t_1 and t_2 if $t \preceq_{\mathcal{E}} t_1$ and $t \preceq_{\mathcal{E}} t_2$. It is a *higher-order pattern generalization* if additionally t is a higher-order pattern. It is the *least general generalization* (lgg in short), aka a *most specific anti-instance,* of t_1 and t_2, if there is no generalization s of t_1 and t_2 which satisfies $t \prec_{\mathcal{E}} s$. An *anti-unification problem* (shortly AUP) is a triple $X(\vec{x}) : t \triangleq s$

where

- $\lambda \overrightarrow{x}.X(\overrightarrow{x})$, $\lambda \overrightarrow{x}.t$, and $\lambda \overrightarrow{x}.s$ are terms of the same type,
- t and s are in η-long β-normal form, and
- X does not occur in t and s.

The variable X is called a *generalization variable.* The term $X(\overrightarrow{x})$ is called the *generalization term.* The variables that belong to $\overrightarrow{x}$, as well as bound variables, are written in the lower case letters $x, y, z, \ldots$. Originally free variables, including the generalization variables, are written with the capital letters $X, Y, Z, \ldots$. This notation intuitively corresponds to the usual convention about syntactically distinguishing bound and free variables. The size of a set of AUPs is defined as $|\{X_1(\overrightarrow{x_1}) : t_1 \triangleq s_1, \ldots, X_n(\overrightarrow{x_n}) : t_n \triangleq s_n\}| = \sum_{i=1}^{n} |t_i| + |s_i|$. Notice that the size of $X_i(\overrightarrow{x_i})$ is not considered. An *anti-unifier* of an AUP $X(\overrightarrow{x}) : t \triangleq s$ is a substitution σ such that $\mathrm{Dom}(\sigma) = \{X\}$ and $\lambda \overrightarrow{x}.X(\overrightarrow{x})\sigma$ is a term which generalizes both $\lambda \overrightarrow{x}.t$ and $\lambda \overrightarrow{x}.s$.

An anti-unifier of $X(\overrightarrow{x}) : t \triangleq s$ is *least general* (or *most specific*) modulo an equational theory $\mathcal{E}$ if there is no anti-unifier ϑ of the same problem that satisfies $\sigma \prec_{\mathcal{E}} \vartheta$. Obviously, if σ is a least general anti-unifier of an AUP $X(\overrightarrow{x}) : t \triangleq s$, then $\lambda \overrightarrow{x}.X(\overrightarrow{x})\sigma$ is a lgg of $\lambda \overrightarrow{x}.t$ and $\lambda \overrightarrow{x}.s$.

Here we consider a variant of higher-order equational anti-unification problem:

Given: Higher-order terms t and s of the same type in η-long β-normal form and an equational theory $\mathcal{E} \subseteq \{\mathsf{A}, \mathsf{C}\}$.

Find: A higher-order pattern generalization r of t and s modulo $\mathcal{E} \subseteq \{\mathsf{A}, \mathsf{C}\}$.

Essentially, we are looking for r which is least general among all higher-order patterns which generalize t and s (modulo $\mathcal{E}$). There can still exist a term which is less general than r, generalizes both s and t, but is not a higher-order pattern. In [8] there is an instance for syntactic anti-unification: if $t = \lambda x, y.f(h(x, x, y), h(x, y, y))$ and $s = \lambda x, y.f(g(x, x, y), g(x, y, y))$, then $r = \lambda x, y.f(Y_1(x, y), Y_2(x, y))$ is a higher-order pattern, which is an lgg of t and s. However, the term $\lambda x, y.f(Z(x, x, y), Z(x, y, y))$, which is not a higher-order pattern, is less general than r and generalizes t and s.

Another important distinguishing feature of higher-order pattern generalization modulo $\mathcal{E}$ is that there may be more than one least general pattern generalization (lgpg) for a given pair of terms. In the syntactic case there is a unique lgpg. The main contribution of this paper is to find conditions on the AUPs under which there is a unique lgpg for equational cases, and introduce weaker-optimality conditions which allow one to greedily search the space for a less general generalization compared to the syntactic one. We formalize these concepts in the following sections.

3   Higher Order Pattern Generalization in the Empty Theory

Below we assume that in the AUPs of the form $X(\overrightarrow{x}) : t \triangleq s$ and the term $\lambda \overrightarrow{x}.X(\overrightarrow{x})$ is a higher-order pattern. We now introduce the rules for the higher-order pattern generalization algorithm from [8], which works for $\mathcal{E} = \emptyset$. It produces syntactic higher-order pattern generalizations in linear time and will play a key role in our optimality conditions introduced in later sections.

These rules work on triples $A; S; \sigma$, which are called *states.* Here A is a set of AUPs of the form $\{X_1(\overrightarrow{x_1}) : t_1 \triangleq s_1, \ldots, X_n(\overrightarrow{x_n}) : t_n \triangleq s_n\}$ that are pending to anti-unify, S is a set of already solved AUPs (the *store*), and σ is a substitution (computed so far) mapping variables to patterns. The symbol $\uplus$ denotes disjoint union.

Dec: **Decomposition**

$$\{X(\overrightarrow{x}) : h(t_1, \ldots, t_m) \triangleq h(s_1, \ldots, s_m)\} \uplus A; \ S; \ \sigma \Longrightarrow$$
$$\{Y_1(\overrightarrow{x}) : t_1 \triangleq s_1, \ldots, Y_m(\overrightarrow{x}) : t_m \triangleq s_m\} \cup A; \ S; \ \sigma\{X \mapsto \lambda\overrightarrow{x}.h(Y_1(\overrightarrow{x}), \ldots, Y_m(\overrightarrow{x}))\},$$

where h is a free constant or $h \in \overrightarrow{x}$, and $Y_1, \ldots, Y_m$ are fresh variables of the appropriate types.

Abs: **Abstraction Rule**

$$\{\{X(\overrightarrow{x}) : \lambda y.t \triangleq \lambda z.s\}\} \uplus A; \ S; \ \sigma \Longrightarrow$$
$$\{X'(\overrightarrow{x}, y) : t \triangleq s\{z \mapsto y\}\} \cup A; \ S; \ \sigma\{X \mapsto \lambda\overrightarrow{x}, y.X'(\overrightarrow{x}, y)\},$$

where X' is a fresh variable of the appropriate type.

Sol: **Solve Rule**

$$\{X(\overrightarrow{x}) : t \triangleq s\} \uplus A; \ S; \ \sigma \Longrightarrow A; \ \{Y(\overrightarrow{y}) : t \triangleq s\} \cup S; \ \sigma\{X \mapsto \lambda\overrightarrow{x}.Y(\overrightarrow{y})\}$$

where t and s are of a basic type, $head(t) \neq head(s)$ or $head(t) = head(s) = Z \notin \overrightarrow{x}$. The sequence $\overrightarrow{y}$ is a subsequence of $\overrightarrow{x}$ consisting of the variables that appear freely in t or in s, and Y is a fresh variable of the appropriate type.

Mer: **Merge Rule**

$$A; \ \{X(\overrightarrow{x}) : t_1 \triangleq t_2, Y(\overrightarrow{y}) : s_1 \triangleq s_2\} \uplus S; \ \sigma \Longrightarrow$$
$$A; \ \{X(\overrightarrow{x}) : t_1 \triangleq t_2\} \cup S; \ \sigma\{Y \mapsto \lambda\overrightarrow{y}.X(\overrightarrow{x}\pi)\}$$

Where $\pi : \{\overrightarrow{x}\} \to \{\overrightarrow{y}\}$ is a bijection, extended as a substitution with $t_1\pi = s_1$ and $t_2\pi = s_2$. Note that in the case of the equational theory we will consider later we would use $=_\varepsilon$ instead of $=$.

We will refer to these generalization rules as $\mathcal{G}_{base}$. To compute generalizations for two simply typed lambda-terms in η-long β-normal form t and s, the algorithm from [8] starts with the *initial state* $\{X : t \triangleq s\}; \emptyset; \emptyset$, where X is a fresh variable, and applies these rules as long as possible. The computed result is the instance of X under the final substitution. It is the syntactic least general higher-order pattern generalization of t and s, and is computed in linear time in the size of the input.

We will use this linear time procedure in the following section to obtain "optimal" least general higher-order pattern generalizations of terms modulo an equation theory. These optimal generalizations are dependent on the generalizations the syntactic algorithm produces. When we need to check more than one decomposition of a given AUP in order to compute the optimal generalizations modulo an equational theory, we compute the optimal generalization for each decomposition path and than compare the results. The details are explained below.

We assume that terms are written in *flattened form*, obtained by replacing all subterms of the form $f(t_1, \ldots, f(s_1, \ldots, s_m), \ldots t_n)$ by $f(t_1, \ldots, s_1, \ldots, s_m, \ldots t_n)$, where $\mathsf{A} \in Ax(f)$. Also, by convention, the term $f(t)$ stands for t, if $\mathsf{A} \in Ax(f)$.

4 Equational Decomposition Rules

In this section we discuss an extension of the basic rules concerning higher-order pattern generalization by decomposition rules for A, C, and AC function symbols. Here, we consider the general, unrestricted case. Efficient special fragments are discussed in the subsequent section.

We start from decomposition rules for associative generalization:

Dec-A-L: **Associative Decomposition Left**

$$\{X(\overrightarrow{x}) : f(t_1, \ldots, t_n) \triangleq f(s_1, \ldots, s_m)\} \uplus A; \ S; \ \sigma \Longrightarrow$$
$$\{Y_1(\overrightarrow{x}) : f(t_1, \ldots, t_k) \triangleq s_1, \ Y_2(\overrightarrow{x}) : f(t_{k+1}, \ldots, t_n) \triangleq f(s_2, \ldots, s_m)\} \cup A;$$
$$S; \ \sigma\{X \mapsto \lambda\overrightarrow{x}.f(Y_1(\overrightarrow{x}), Y_2(\overrightarrow{x}))\},$$

where $Ax(f) = \{\mathsf{A}\}$, $1 \le k \le n-1$, $n, m \ge 2$, and Y_1 and Y_2 are fresh variables of appropriate types.

Dec-A-R: **Associative Decomposition Right**

$$\{X(\overrightarrow{x}) : f(t_1, \ldots, t_n) \triangleq f(s_1, \ldots, s_m)\} \uplus A; \ S; \ \sigma \Longrightarrow$$
$$\{Y_1(\overrightarrow{x}) : t_1 \triangleq f(s_1, \ldots, s_k), Y_2(\overrightarrow{x}) : f(t_2, \ldots, t_n) \triangleq f(s_{k+1}, \ldots, s_m)\} \cup A;$$
$$S; \ \sigma\{X \mapsto \lambda\overrightarrow{x}.f(Y_1(\overrightarrow{x}), Y_2(\overrightarrow{x}))\},$$

where $Ax(f) = \{\mathsf{A}\}$, $1 \le k \le m-1$, $n, m \ge 2$, and Y_1 and Y_2 are fresh variables of appropriate types.

We refer to the extension of $\mathcal{G}_{base}$ by the above associativity rules as $\mathcal{G}_A$ and extend the termination, soundness and completeness results for $\mathcal{G}_{base}$ to $\mathcal{G}_A$.

▶ **Theorem 1** (Termination). *The set of transformations $\mathcal{G}_A$ is terminating.*

Proof. Termination follows from the fact that $\mathcal{G}_{base}$ terminates [8] and the rules Dec-A-L and Dec-A-R can be applied finitely many times.   ◀

▶ **Theorem 2** (Soundness). *If $\{X : t \triangleq s\}; \emptyset; \emptyset \Longrightarrow^* \emptyset; S; \sigma$ is a transformation sequence of $\mathcal{G}_A$, then $X\sigma$ is a higher-order pattern in η-long β-normal form and $X\sigma \preceq t$ and $X\sigma \preceq s$.*

Proof. It was shown in [8] that $\mathcal{G}_{base}$ is sound. Let us assume as a base case that all occurrences of associative function symbols in $t \triangleq s$ have two arguments. Then the rules Dec-A-L and Dec-A-R are equivalent to the *Dec* rule. As an induction hypothesis (IH), assume soundness holds when all occurrences of associative function symbols in $t \triangleq s$ have $\le n$ arguments. We show that it holds for $n + 1$. Let $t \triangleq s$ be of the form $f(t_1, \ldots, t_m) \triangleq f(s_1, \ldots, s_k)$ for $\max\{k, m\} \le (n + 1)$ and let associative function symbols occurring in $t_1, \ldots t_m, s_1, \ldots s_k$ have at most n arguments. Any application of Dec-A-L or Dec-A-R will produce two AUPs for which the IH holds, and thus, the theorem holds. We can extend this argument to an arbitrary number of associative function symbols with $n + 1$ arguments with another induction.   ◀

▶ **Theorem 3** (Completeness). *Let $\lambda\overrightarrow{x}.t_1$ and $\lambda\overrightarrow{x}.t_2$ be higher-order terms and $\lambda\overrightarrow{x}.s$ be a higher-order pattern such that $\lambda\overrightarrow{x}.s$ is a generalization of both $\lambda\overrightarrow{x}.t_1$ and $\lambda\overrightarrow{x}.t_2$ modulo associativity. Then there exists a transformation sequence $\{X(\overrightarrow{x}) : t_1 \triangleq t_2\}; \emptyset; \emptyset \Longrightarrow^* \emptyset; S; \sigma$ in $\mathcal{G}_A$ such that $\lambda\overrightarrow{x}.s \preceq X\sigma$.*

Proof. We can reason similarly to the previous proof. It was shown in [8] that $\mathcal{G}_{base}$ is complete. Let us assume as a base case that all occurrences of associative function symbols in $t \triangleq s$ have two arguments. Then the rules Dec-A-L and Dec-A-R are equivalent to the *Dec* rule and completeness holds. When we have $n + 1$ arguments there are n ways to group the arguments associatively and the decompositions rules Dec-A-L and Dec-A-R allow one to consider all groupings.   ◀

The addition of associative function symbols allows for more than one decomposition and thus more than one lgg in contrast to higher-order pattern generalization which results in a unique lgg . If we wish to compute the complete set of lggs we would simply exhaust all possible applications of the above rules. However, for most applications an "optimal" generalization is sufficient. We postpone discussion till the next section.

The decomposition rule for commutative symbols is also pretty intuitive:

Dec-C: **Commutative Decomposition**

$$\{X(\overrightarrow{x}) : f(t_1, t_2) \triangleq f(s_1, s_2)\} \uplus A; \ S; \ \sigma \Longrightarrow$$
$$\{Y_1(\overrightarrow{x}) : t_1 \triangleq s_i, \ Y_2(\overrightarrow{x}) : t_2 \triangleq s_{(i \mod 2)+1}\} \cup A; \ S; \ \sigma\{X \mapsto \lambda \overrightarrow{x}.f(Y_1(\overrightarrow{x}), Y_2(\overrightarrow{x}))\},$$

where $Ax(f) = \{\mathsf{C}\}$, $i \in \{1, 2\}$, and Y_1 and Y_2 are fresh variables of appropriate types.

We refer to the extension of $\mathcal{G}_{base}$ by the commutativity rule as $\mathcal{G}_C$. We can easily extend the termination, soundness, and completeness results to $\mathcal{G}_{\mathsf{C}}$. Notice that also for commutative generalization, the lgg is not necessarily unique.

Unlike commutativity, which considers a fixed number of terms, and associativity, which enforces an ordering on terms, AC function symbols allow an arbitrary number of arguments with no fixed ordering on the terms. The corresponding decomposition rules take it into account:

Dec-AC-L: **Associative-Commutative Decomposition Left**

$$\{X(\overrightarrow{x}) : f(t_1, \ldots, t_n) \triangleq f(s_1, \ldots, s_m)\} \uplus A; \ S; \ \sigma \Longrightarrow$$
$$\{Y_1(\overrightarrow{x}) : f(t_{i_1}, \ldots, t_{i_l}) \triangleq s_k, \ Y_2(\overrightarrow{x}) : f(t_{i_{(l+1)}}, \ldots, t_{i_n})$$
$$\triangleq f(s_1, \ldots, s_{k-1}, s_{k+1}, \ldots, s_m)\} \cup A;$$
$$S; \ \sigma\{X \mapsto \lambda \overrightarrow{x}.f(Y_1(\overrightarrow{x}), Y_2(\overrightarrow{x}))\},$$

where $Ax(f) = \{\mathsf{A}, \mathsf{C}\}$, $\{i_1, \ldots, i_n\} \equiv \{1, \ldots, n\}$, $l \in \{1, \ldots, n-1\}$, $k \in \{1, \ldots, m\}$, $n, m \geq 2$, and Y_1 and Y_2 are fresh variables of appropriate types.

Dec-AC-R: **Associative-Commutative Decomposition Right**

$$\{X(\overrightarrow{x}) : f(t_1, \ldots, t_n) \triangleq f(s_1, \ldots, s_m)\} \uplus A; \ S; \ \sigma \Longrightarrow$$
$$\{Y_1(\overrightarrow{x}) : t_k \triangleq f(s_{i_1}, \ldots, s_{i_l}), \ Y_2(\overrightarrow{x}) : f(t_1, \ldots, t_{k-1}, t_{k+1}, \ldots, t_n)$$
$$\triangleq f(s_{i_{(l+1)}}, \ldots, s_{i_m})\} \cup A;$$
$$S; \ \sigma\{X \mapsto \lambda \overrightarrow{x}.f(Y_1(\overrightarrow{x}), Y_2(\overrightarrow{x}))\},$$

where $Ax(f) = \{\mathsf{A}, \mathsf{C}\}$, $\{i_1, \ldots, i_m\} \equiv \{1, \ldots, m\}$, $l \in \{1, \ldots, m-1\}$, $k \in \{1, \ldots, n\}$, $n, m \geq 2$, and Y_1 and Y_2 are fresh variables of appropriate types.

We refer to the extension of $\mathcal{G}_{base}$ by the AC decomposition rules as $\mathcal{G}_{\mathsf{AC}}$. Again, termination, soundness and completeness are easily extended to this case.

5 Towards Special Fragments

This section is devoted to computing special kind of "optimal" generalizations, which can be done more efficiently than the general unrestricted cases considered in the previous section.

The idea is the following: The equational decomposition rules introduce branching in the search space. Each branch can be developed in linear time, but there can be too many of them. However, if the branching factor is bounded, we could choose one of the alternative states (produced by decomposition) based on some "optimality" criterion, and develop only that branch. Such a greedy approach will give one "optimal" generalization.

In order to have a "reasonable" complexity, we should be able to choose such an optimal state from "reasonably" many alternatives in "reasonable" time. For this, our idea is to treat all the alternative states obtained by an equational decomposition step as syntactic anti-unification problems, compute lggs for each of them (which can be done in linear time), choose the best one among those lggs (e.g., less general than the others, or, if there are several such results, use some heuristics), and restart equational anti-unification algorithm from the state which led to the computation of that best syntactic lgg. When the branching factor is constant, this leads to a quadratic algorithm, and when it is linearly bounded, we get a cubic algorithm. These are the cases we consider below. We would also need to decompose in a more clever way than in the rules above, where the decomposition was based on an arbitrary choice of a subterm.

Hence, we need to identify fragments of equational anti-unification problems which would have the decomposition branching factor constant or linearly bounded. We start by introducing the following concepts.

▶ **Definition 4** (*E*-refined generalization). Given two terms t and s and their $\mathcal{E}$-generalizations r and r', we say that r is *at least as good as* r' with respect to $\mathcal{E}$ if either $r' \preceq_{\mathcal{E}} r$ or they are not comparable with respect to $\preceq_{\mathcal{E}}$.

An $\mathcal{E}$-generalization r of t and s is called their *E-refined generalization* iff r is at least as good (with respect to $\mathcal{E}$) as a syntactic lgg of t and s.

Note that every syntactic generalization is also an $\mathcal{E}$-generalization. A direct consequence of this definition is that every element of the minimal complete set of $\mathcal{E}$-generalizations (where $\mathcal{E}$ is A, C, or AC) of two terms is an $\mathcal{E}$-refined generalization of t and s. However, there might exist $\mathcal{E}$-refined generalizations which do not belong to the minimal complete set of generalizations.

Looking back at the informal description of the construction above, we can say that at each branching point we will be aiming at choosing the alternative that would lead to "the best" $\mathcal{E}$-refined generalization.

The concept of *E*-refined allows us to compute better generalizations than the base procedure would do, without concerning ourselves with certain difficult to handle decompositions. We will outline what we mean by "difficult" in later sections. Some of these difficult decompositions can be handled by finding *alignments* between two sequences of terms.

▶ **Definition 5** (Alignment, Rigidity Function). Let w_1 and w_2 be strings of symbols. Then the sequence $a_1[i_1, j_1] \cdots a_n[i_n, j_n]$, for $n \geq 0$ and a_k are not variables, is an *alignment* if
- i's and j's are integers such that $0 < i_1 < \cdots < i_n < |w_1|$ and $0 < j_1 < \cdots < j_n < |w_2|$, and
- $a_k = w_1|_{i_k} = w_2|_{j_k}$, for all $1 \leq k \leq n$. An alignment of the form $a_1[i, j]$ will be referred to as a *singleton alignment*, where $t|_{\alpha}$ denote the subterm at position α.

The set of all alignments will be denoted by **A**. A *(singleton) rigidity function* $\mathcal{R}$ is a function that returns, for every pair of strings of symbols w_1 and w_2, a set of (singleton) alignments of w_1 and w_2.

The main intuition behind the use of rigidity functions for generalization is to capture the structure (modulo a given rigidity property) of as many nonvariable terms as possible.

▶ **Definition 6** (Pair of argument head sequences and multisets). Let $t = f(t_1, \ldots, t_n)$ and $s = f(s_1, \ldots, s_m)$. Then the *pair of argument head sequences* and the *pair of argument head*

multisets of t and s, denoted respectively as $pahs(t, s)$ and $pahm(t, s)$, are defined as follows:

$$pahs(t, s) = \langle (head(t_1), \ldots, head(t_n)),\ (head(s_1), \ldots, head(s_m)) \rangle .$$
$$pahm(t, s) = \langle \{\!\{head(t_1), \ldots, head(t_n)\}\!\},\ \{\!\{head(s_1), \ldots, head(s_m)\}\!\} \rangle .^2$$

These notions extend to AUPs: A pair of argument head sequences (resp. multisets) of an AUP $X(\vec{x}) : t \triangleq s$ is the pair of argument head sequences (resp. multisets) of the terms t and s.

There is a subset of AUPs, referred to as 1-*Determined AUPs*, which contain associative function symbols and have interesting $\mathcal{E}$-refined generalizations are computable in linear time. The more general r-determined AUPs allow a bounded number of possible choices, that is r choices, whenever associative decomposition may be applied. Even for 2-determined AUPs computing the set of lggs is of exponential complexity. Therefore, we introduce the notion of $(\mathcal{R}, C, \mathcal{G})$-*optimal generalization* where $\mathcal{R}$ is a so called rigidity function [11] and C is a choice function picking one of available decompositions. Under such optimality conditions, we are able to compute an $\mathcal{E}$-refined generalization in quadratic time for k-determined AUPs and in cubic time for arbitrary AUPs with associative function symbols.

The equational decomposition rules above are too non-deterministic and the computed set of generalizations has to be minimized to obtain minimal complete sets of generalizations. However, even if we performed more guided decompositions, obtaining e.g., terms with the same head in new AUPs (as in [11]), there would still be alternatives. For instance, consider the following AUP where f is associative: $X(\vec{x}) : f(t_1, \ldots t_i, \ldots, t_j, \ldots, t_n) \triangleq f(s_1, \ldots s_i, \ldots, s_j, \ldots, s_m)$. Now let $head(t_i) = head(s_j)$, $head(s_i) = head(t_j)$, and for every other term comparison whose index is $\leq j$ the head symbols are not equivalent. Under these assumptions there is not enough information to decide which decomposition is less general. Furthermore, this can be generalized from two possible decompositions to k possibilities.

Under certain conditions we can force a term to have a single decomposition path, what we will refer to as a 1-*determined* condition which is equivalent to unique longest common subsequence of head symbols. We formally define k-determined AUPs using the following sequence of definitions:

▶ **Definition 7** (k-determinate set)**.** Given the pair of sequences of symbols $\langle s_1, s_2 \rangle$ with $s_1 = (a_1, \ldots, a_n)$ and $s_2 = (b_1, \ldots, b_m)$, and a positive integer k, the *(strict) k-determinate set* of s_1 and s_2, denoted $det\,(k, s_1, s_2)\ (det_s\,(k, s_1, s_2))$, is defined as follows:

- If $n = 0$ and $m \neq 0$ or vice versa, then $det\,(k, s_1, s_2) = \emptyset$.
- Otherwise, let $1 \leq i \leq \min(n, m)$ be a number such that for the multiset $M_i = (\{\!\{a_1\}\!\} \cap \{\!\{b_1\}\!\}) \cup (\{\!\{a_2, \ldots, a_i\}\!\} \cap \{\!\{b_2, \ldots, b_i\}\!\}) \neq \emptyset$ we have $M_i \cap \{\!\{b_{i+1}, \ldots, b_m\}\!\} = M_i \cap \{\!\{a_{i+1}, \ldots, a_n\}\!\} = \emptyset$. Let $K\ (K_s)$ be the set of pairs $\{a_{j_1}[j_1, j_2] \mid a_{j_1} = b_{j_2}$ and $j_1 = 1$ iff $j_2 = 1\}$ $(\ \{a_{j_1}[j_1, j_2] \mid a_{j_1} = b_{j_2}\})$. If K has at most k elements, then

$$det(k, s_1, s_2) := \bigcup_{a_{j_1}[j_1, j_2] \in K} add(a_{j_1}[j_1, j_2], det\,(k, (a_{j_1+1}, \ldots, a_n), (b_{j_2+1}, \ldots, b_m))).$$

$$add(a, A) = \left\{ \begin{array}{c|c} \{(a, A)\} & A \neq \emptyset \\ \emptyset & otherwise \end{array} \right.$$

- Otherwise, $det(k, s_1, s_2) = \{\emptyset\}$.

---

2 $\{\!\{\circ\}\!\}$ denotes a multiset.

Note that $det_s\,(k, s_1, s_2)$ is defined analogously using K_s instead of K. We will refer to the pairs (a, A) where a is a singleton alignment and A a k-determinate set as *blocks*.

We will use $det_s\,(k, s_1, s_2)$ when considering commutativity in Section 7.

▶ **Example 8.** We illustrate the previous definition:

- $det\,(1, (a, b), (a, b)) = \{(a[1, 1]\;;\;\{(b[1, 1]\;;\;\{\emptyset\})\})\}.$
- $det\,(1, (a, a), (b, a)) = \{(\{a[2, 2]\;;\;\{\emptyset\})\}.$
- $det\,(1, (a, c, c, b, a, c), (a, d, b, a, c)) = \{(a[1, 1]\;;\;\{(b[3, 2]\;;\;\{(a[1, 1]\;;\;\{(c[1, 1]\;;\;\{\emptyset\})\})\})\})\}.$
- $det\,(1, (a, b, a), (c, a, c, b)) = \{\emptyset\}$
- $det\,(1, (a, b, d), (c, a, b, c)) = \{(b[2, 3]\;;\;\{\emptyset\})\}$
- $det\,(2, (a, b, a), (c, a, b, c)) = \{(b[2, 3]\;;\;\{\emptyset\})\}$
- $det_s\,(1, (a, b), (b, a)) = \{(a[1, 2]\;;\;\{\emptyset\})\,,(b[2, 1]\;;\;\{\emptyset\})\}$
- $det\,(2, (c, a, b, c), (d, b, a, d)) = \{(a[2, 3]\;;\;\{\emptyset\})\,,(b[3, 2]\;;\;\{\emptyset\})\}.$
- $det\,(3, (a, b, a, c, d), (c, a, b, a, d)) =$
  $\{(b[2, 3]\;;\;\{(a[1, 1]\;;\;\{\emptyset\})\})\,,(a[3, 2]\;;\;\{(d[2, 3]\;;\;\{\emptyset\})\})\,,(a[3, 4]\;;\;\{\emptyset\})\}.$
- $det\,(k, (a, a), (b, c, d)) = \{\emptyset\}.$
- $det\,(k, (a, b), (a)) = \emptyset.$
- $det\,(k, (a, a), (a)) = \{\emptyset\}.$

Even though $det\,(k, (a, b), (a))$ and $det\,(k, (a, a), (a))$ are related the formalism does not handle them as similar. This merely makes the formalism a little more restricted. Notice that a unique longest common subsequence of two symbol sequences is not equivalent to k-determined. Consider the following example:

- $det\,(k, (c, a, a, d), (c, a, b, a, d)) = \{(c[1, 1]\;;\;\{(a[1, 1]\;;\;\{(d[2, 3]\;;\;\{\emptyset\})\})\})\}.$

The alignment representing its longest common subsequence is

- $c[1, 1]a[2, 2]a[3, 4]d[4, 5]$

▶ **Definition 9** (k-determined term pairs). A pair of terms $\langle t, s \rangle$ is k-determined iff either $head(t) \neq head(s)$ or $head(t) = head(s) = f$ and $Ax(f) = \emptyset$, or $Ax(f) = \{\mathsf{A}\}$ and $det\,(k, pahs(t, s)) \neq \emptyset$. Furthermore we say that the pair $\langle t, s \rangle$ is *total k-determined* if $t = \lambda x_1, \ldots, x_n.t'$, $s = \lambda y_1, \ldots, y_n.s'$ and t' and s' are η-equivalent to t'' and s'' with $|t''| = |s''| = 1$, or for each $(a[i, j], S) \in det\,(k, pahs(t, s))$ where t_i is the term at the ith position of t and s_j is the term at the jth position of s the term pair $\langle t_i, s_j \rangle$ is total k-determined.

▶ **Proposition 1.** *The complexity of checking if the terms of an AUP*

$$X(\bar{x}) : \lambda x_1, \ldots, x_l.f(t_1, \ldots, t_n) \triangleq \lambda y_1, \ldots, y_k.f(s_1, \ldots, s_m)$$

is 1-determined is $O(n)$ and total 1-determined is $O(n^2)$, where n is maximum of the length of the two terms.

Checking k-determinedness of an AUP is a harder problem complexity-wise. For example, given the sequences $(a, \ldots, a)$ and $(a, \ldots, a)$ there are n^2 ways to align the terms which have to be checked. Moreover, if we want to check total k-determinedness we have to again do a quadratic check for each pair of aligned terms resulting in an $O(n^4)$ procedure.

6   Associative Generalization: Special Fragments and Optimality

6.1   Associativity and 1-Determined AUPs

We provide a linear time algorithm for higher-order $\{A\}$-refined pattern generalization of AUPs which are 1-determined. Essentially, at every step there is a single decomposition choice which can be made.

▶ **Theorem 10.** *A higher-order* $\{A\}$*-refined pattern generalizer for a* total 1-determined *AUP can be computed in linear time.*

Proof. If the AUP does not contain an associative function symbol, then its E-refined generalization, which is also an lgg, can be computed in linear time [8]. If it does contain an associative function symbol, we have two alternatives: either every occurrence of the associative function symbol has two arguments (remember that our terms are in flattened form), or not. In the former case, the associative decomposition rules do not differ from the syntactic decomposition rule Dec and we can only apply the latter. It means that we can still use the linear algorithm from [8]. The rest of the proof is about the case when there are occurrences of associative function symbols with more than two arguments. The proof goes by induction on the maximal number of such arguments.

We assume for the induction hypothesis that if every instance of the associative function symbol in the AUP has at most n arguments, then it is solvable in linear time, and show that the same holds for $n + 1$. Let us assume that the AUP we are currently considering has the following form $X(\vec{x}) : f(t_1, \ldots, t_m) \triangleq f(s_1, \ldots, s_k)$ where f is associative and $\max\{m, k\} = n + 1$. Assume without loss of generality that $k = n + 1$. Also, assume that no other occurrence of f in the given AUP has more than n arguments. We make this assumption in order to reduce the complexity of associative decomposition in the AUP and thus, apply the induction hypothesis. If $head(t_1) = head(s_1)$,then their lgg should not be a variable. Therefore, we can apply Dec-A-L, which results in the AUPs $X(\vec{x}) : t_1 \triangleq s_1$ (whose further decomposition will make sure that they t_1 and s_1 are not generalized by a generalization variable) and $X(\vec{x}) : f(t_2, \ldots, t_m) \triangleq f(s_2, \ldots, s_{n+1})$. Notice that both of the resulting AUPs, by our assumptions, only contain f with not more than n arguments. Thus, by the induction hypothesis the theorem holds in this case.

For the next step we assume s and t are the terms of the AUP and that $(h[l, l], S) \in det(1, pahs(t, s))$ s.t. $Ax(h) = \{A\}$. Therefore, we can perform Dec-A-L only on the first argument $l - 1$ times, which gives the following new AUPs: $\{X_1(\vec{x}) : t_1 \triangleq s_1, \ldots, X_{l-1}(\vec{x}) : t_{l-1} \triangleq s_{l-1}, X_l(\vec{x}) : f(t_l \ldots, t_m) \triangleq f(s_l, \ldots, s_{n+1})\}$. All the resulting AUPs, by our assumptions, only contain f with not more than n arguments, thus by the induction hypothesis the theorem holds in this case.

For the next step we assume s and t are the terms of the AUP and that $(h[i, j], S) \in det(1, pahs(t, s))$ s.t. $Ax(h) = \{A\}$ and $i \neq j$. This is similar to the previous case except there is more than one possible way to apply associative decomposition. More precisely, the number of possible ways is $F(l - j + 1)$ where

$$F(0) = 1, \qquad F(r + 1) = \sum_{w=1}^{r+1} F(r + 1 - w) \quad \text{for } r \geq 0.$$

which is roughly $F(r) = 2^{(r-1)}$. Note that $F(\cdot)$ is derived from the combinatorics of the associative decomposition rule and concerns the number of possible pairings with respect to 1-determinacy. However, being that none of the head symbols of obtained term-pairs are

equivalent nor can their head symbols be equivalent to f, we know that none of the resulting AUPs will require further decomposition. Thus, we need to apply associative decomposition. This can be easily performed be performed by some heuristic. The result will be a set of AUPs containing $X(\vec{x}) : f(t_j \ldots t_m) \triangleq f(s_l, \ldots s_{n+1})$ and thus by the induction hypothesis and our assumptions, the theorem holds.

For the final step we just need to apply a simple induction argument on the number of times in a term the associative symbol f occurs with arity $n + 1$. The above argument provides the step case and base case being that we prove the theorem for one occurrence and can use the proof for p occurrences. Thus, the theorem holds.   ◀

In the next section we consider AUPs which are k-determined for $k > 1$. This will requires a new concept of optimality based on a choice function greedily applied during decomposition.

6.2   Choice Functions and Optimality

In this section procedures and optimality conditions for total k-determined AUPs, for $k > 1$, that is AUPs where there are at most k ways to apply equational decomposition.

If we were to compute the set of E-refined generalizations for a total k-determined AUP by testing every decomposition, even for $k = 2$ the size of search space is too large to deal with efficiently. However, we can find a $(\mathcal{R}, C, \mathcal{G})$-*optimal E-refined generalization* (precisely defined below) in quadratic time, where $\mathcal{R}$ is a singleton rigidity function, C a $\mathcal{R}$-choice function, $\mathcal{G}$ is a set of state transformation rules. Essentially, $(\mathcal{R}, C, \mathcal{G})$-optimality means the $\mathcal{R}$-choice function chooses the "right" computation path via $\mathcal{G}$ based on the singleton rigidity function $\mathcal{R}$. The effect is that we reduce the problem of total k-determined AUPs to the case of total 1-determined AUPs with the additional complexity of computing the choice function at each step. We will provide a choice function with linear time complexity based on the procedure for $\mathcal{G}_{base}$.

We will denote the set of all AUPs by $\mathbb{A}$. We will need the concept for the following definitions.

▶ **Definition 11** ((P, a)-decomposition). Let $P \equiv X(\bar{x}) : \lambda x_1, \ldots, x_l.f(t_1, \ldots, t_n) \triangleq \lambda y_1, \ldots, y_k.f(s_1, \ldots, s_m)$, a is an alignment of $\langle w_1, w_2 \rangle_P$ (see Definition 6). An (P, a)-*decomposition of P* is $dec(P, a) = \{Y_{(i,j)}(\vec{y}_{(i,j)}) : t_i \triangleq s_j \mid h[i, j] \in a \}$, where $Y_{(i,j)}$ are new variables of appropriate type and $\vec{y}_{(i,j)}$ are bound variables from $\vec{x}$, which appear in $t_i \triangleq s_j$.

▶ **Definition 12** ($\mathcal{G}$-feasible). Let $A; S; \sigma$ be a state s.t. $P \in A$ where $P \equiv X(\vec{x}) : \lambda x_1, \ldots, x_l.f(t_1, \ldots, t_n) \triangleq \lambda y_1, \ldots, y_k.f(s_1, \ldots, s_m)$, a be an alignment of $\langle w_1, w_2 \rangle_P$ and $\mathcal{G}_{base} \subseteq \mathcal{G}$ be a set of state transformation rules. We say that $dec(P, a)$ is $\mathcal{G}$-*feasible* if there exists $A; S; \sigma \Longrightarrow^* A'; S'; \sigma'$ using $\mathcal{G}$ such that $A' = (A \setminus P) \cup dec(P, a)$.

▶ **Definition 13** ($(\mathcal{R}, P, \mathcal{G})$-branching). Let $P \equiv X(\vec{x}) : \lambda x_1, \ldots, x_l.f(t_1, \ldots, t_n) \triangleq \lambda y_1, \ldots, y_k.f(s_1, \ldots, s_m)$, $\langle w_1, w_2 \rangle_P$ be its pair of argument head sequences, $\mathcal{R}$ be a singleton rigidity function, and $\mathcal{G}_{base} \subseteq \mathcal{G}$ be a set of state transformation rules. An $(\mathcal{R}, P, \mathcal{G})$-*branching* is a set $B(\mathcal{R}, P) = \{dec(P, a) \mid a \in \mathcal{R}(w_1, w_2)$ and $dec(P, a)$ is $\mathcal{G}$-feasible$\}$.

▶ **Definition 14** ($\mathcal{R}$-Choice function). Let $\mathcal{R}$ be a singleton rigidity function and $\mathcal{G}_{base} \subseteq \mathcal{G}$ be a set of state transformation rules. An $\mathcal{R}$-choice function $C_{(\mathcal{R}, \mathcal{G})} : \mathbb{A} \to \mathbf{A}$ is a partial function from AUPs to alignments such that if for some $P \in \mathbb{A}$, $C_{(\mathcal{R}, \mathcal{G})}(P) = a$, then $dec(P, a) \in B(\mathcal{R}, P)$.

▶ **Definition 15** ($(\mathcal{R}, C, \mathcal{G})$-optimal generalization). Let A be $\{X(\bar{x}) : t \triangleq s\}$, $\mathcal{R}$ be a singleton rigidity function, C be an $\mathcal{R}$-choice function, and $\mathcal{G}_{base} \subseteq \mathcal{G}$ be a set of state transformation

rules, which compute generalizations. We say that a generalization k of the terms t and s is an $(\mathcal{R}, C, \mathcal{G})$-optimal generalization if $r = X\sigma$, where σ is resulting from the derivation $A; \emptyset; \emptyset \Longrightarrow^* \emptyset; S; \sigma$ using the rules of $\mathcal{G}$, in which every decomposition is either syntactic or respects C-equivalence.

In the following subsection we show how the above definitions can lead to a more general result (compared to the one in the previous section) concerning associative generalization.

6.3 k-Determined Associative Generalization

Before defining our concrete choice function, we must define the singleton rigidity function we will use. Intuitively, it should select alignments from prefixes of involved sequences. The prefixes are of the same length and should be maximal among those that contain at most k common elements. Formally, it is defined as follows:

▶ **Definition 16.** Let $w_1 = (a_1, \ldots, a_n)$ and $w_2 = (b_1, \ldots, b_m)$ be sequences of symbols and $k \geq 1$ be an integer. We define the singleton rigidity function $\mathcal{R}_{\mathsf{A}}^k$ as

$$
\mathcal{R}_{\mathsf{A}}^k(w_1, w_2) = \left\{ \begin{array}{c|c} \{a_l\,[l, k] \mid (a_l\,[l, k], S) \in det\,(k, w_1, w_2)\} & det\,(k, w_1, w_2) \neq \emptyset \\ \emptyset & otherwise \end{array} \right. \tag{1}
$$

Now we define a choice function taking an arbitrary singleton rigidity function.

▶ **Definition 17.** Let $P \equiv X(\vec{x}) : \lambda x_1, \ldots, x_l.f(t_1, \ldots, t_n) \triangleq \lambda y_1, \ldots, y_k.\ f(s_1, \ldots, s_m)$ be an AUP and f a function symbol such that $Ax(f) \not\equiv \emptyset$. We define the choice function $C_{(\mathcal{R}, \mathcal{G})}$, where $\mathcal{R}$ is a singleton rigidity function, and $\mathcal{G}$ is a set of state transformation rules containing $\mathcal{G}_{base}$, as follows:

$$
C_{(\mathcal{R}, \mathcal{G})}(P) = \left\{ \begin{array}{c|c} a_{\min} & B(\mathcal{R}, P) \not\equiv \emptyset \\ \mathbf{undef} & otherwise \end{array} \right. \tag{2}
$$

where $a_{\min}$ is an alignment of $(head(t_1), \ldots, head(t_n))$ and $(head(s_1), \ldots, head(s_m))$ such that

- $dec(P, a_{\min}) \in B(\mathcal{R}, P)$,
- for $dec(P, a) \in B(\mathcal{R}, P)$, let $D(a)$ be the derivation $D(a) = \{P\}; \emptyset; \emptyset \Longrightarrow_{\mathcal{G}}^* dec(P, a); S'; \sigma' \Longrightarrow_{\mathcal{G}_{base}}^* \emptyset; S; \sigma_a$.
  Then for each $a \neq a_{\min}$, the corresponding $D(a)$ computes σ_a such that $X\sigma_a$ is more general than $X\sigma_{a_{\min}}$, where $\sigma_{a_{\min}}$ is computed by $D(a_{\min})$. If there are several such $a_{\min}$'s, $C_{(\mathcal{R}, \mathcal{G})}(P)$ is defined as one of them (chosen by some heuristics).

The choice function outlined above uses the linear time procedure $\mathcal{G}_{base}$ to make a choice between the various possible alignments. Notice that we use associative decomposition for $\{P\}; \emptyset; \emptyset \Longrightarrow^* dec(P, a); S'; \sigma'$ and syntactic decomposition in the derivation $dec(P, a); S'; \sigma' \Longrightarrow^* \emptyset; S; \sigma_a$.

▶ **Theorem 18.** A $(\mathcal{R}_A^k, C_{(\mathcal{R}_A^k, \mathcal{G}_A)}, \mathcal{G}_A)$-*optimal higher-order* $\{\mathsf{A}\}$-*refined pattern generalization for a* total k-determined *AUP* $X(\vec{x}) : t \triangleq s$ *can be computed in* $O(n^2)$ *where n is the size of the AUP.*

Proof. This follows from the existence of a linear algorithm for the computation of lggs using $\mathcal{G}_{base}$ and the linear time algorithm of theorem 10. Note that k is constant and thus does not show up in complexity statement. ◀

6.4   Step Optimal Generalization for Full Associativity

Completely dropping the determinedness restrictions on the AUPs containing associative function symbols is the same as considering $O(n)$-determined AUPs. We have already shown that this problem is naively solvable by an exponential procedure, even when we consider $O(1)$-determined AUPs. In this section we again consider the problem of finding a $(\mathcal{R}_A^{O(n)}, C_{(\mathcal{R}_A^{O(n)}, \mathcal{G}_A)}, \mathcal{G}_A)$-optimal generalization where n in the Landau-notation refers to the maximum number of arguments of any subterms in the given AUP. However, this time the resulting algorithm is cubic in complexity being that r in r–determined is no longer a constant. By $\mathcal{R}_A^{O(n)}$ we mean the singleton rigidity function which instead of looking for an r-determined subsequence just considers the largest feasible multiset intersection.

▶ **Theorem 19.** *A $(\mathcal{R}_A^{O(n)}, C_{(\mathcal{R}_A^{O(n)}, \mathcal{G}_A)}, \mathcal{G}_A)$-optimal higher-order $\{A\}$-refined pattern generalization for an AUP $X(\overrightarrow{x}) : t \triangleq s$ can be computed in $O(n^3)$ time where n is the size the AUP.*

Now that we have completed our analysis of associative function symbols, the simplest of the cases we consider, we move on to the more interesting cases of unit and commutative decomposition as well as the combinations of these algebraic properties.

7   Commutative Case

Notice that in the case of commutative decomposition if all four terms (or three terms) have the same head symbol we end up with similar issues as in the associativity case. We can use strict 2-determined to restrict the considered AUPs.

▶ **Theorem 20.** *A higher-order $\{C\}$-refined pattern generalization, for a total strict 1-determined AUP can be computed in linear time.*

Proof. Similar to the proof of Theorem 10.   ◀

Note that the case $f(t_1, t_2) \triangleq f(s_1, s_2)$, where $head(t_1) = head(s_1)$ and $head(t_2) = head(s_2)$, is considered by the procedure of Theorem 20, but not $f(t_1, t_2) \triangleq f(s_2, s_1)$ This is an issue with the definition of total strict 1-determined. We can fix this problem by performing an addition check to see if a permutation of the terms on the left or right side results in a better alignment. We now present a procedure for full commutativity, that is without restrictions which has a quadratic complexity (see Theorem 18.

▶ **Definition 21.** Let $w_1 = (a_1, \ldots, a_n)$ and $w_2 = (b_1, \ldots, b_m)$ be sequences of symbols and $k \geq 1$ be an integer. We define the rigidity function $\mathcal{R}_C$ returning all alignments.

When the rigidity function $\mathcal{R}_C$ is used all by our procedure there will be at most 4 alignments.

▶ **Corollary 22.** *A $(\mathcal{R}_C, C_{(\mathcal{R}_C, \mathcal{G}_{\{C\}})}, \mathcal{G}_{\{C\}})$-optimal higher-order $\{C\}$-refined pattern generalization for an AUP can be computed in quadratic time.*

8   Associative-Commutative Case

In this section we consider functions f such that $Ax(f) = \{A, C\}$. Unfortunately, when a function is both associative and commutative, the number of possible decomposition paths is even greater than the previously considered cases and thus we need to further restrict the term

structure. To provide a better understanding of why this is the case, consider a k-determined AUP where the multiset intersection is of size $O(k)$ and only contains one function symbol. This implies that there are $O(k^2)$ possible decompositions of the terms in the first multiset intersection of the terms containing k alignments. This is not even considering that there might be more than one function symbol in the AUP. The problem is that the more terms with the same head symbol, the more combinations we must check. Unlike commutativity, which considers a fixed number of terms, and associativity, which enforces an ordering on terms, associative-commutativity allows an arbitrary number of arguments with no fixed ordering on the terms. We can get around this problem by considering special cases of AUPs where arguments of an associative-commutativity symbol have distinct heads.

Unfortunately, the concept of (strict) k-determined AUPs does not lead to a linear algorithm in the case of AC-generalization. Actually, this concept is not even meaningful for such an equational theory, since terms are not ordered in any particular way. Instead, we need to consider so called (k, l)-distinct AUPs, which are defined as follows:

▶ **Definition 23.** Let $P \equiv X(\vec{x}) : \lambda x_1, \ldots, x_l.f(t_1, \ldots, t_n) \triangleq \lambda y_1, \ldots, y_k.\ f(s_1, \ldots, s_m)$, $pahm(f(t_1, \ldots, t_n), f(s_1, \ldots, s_m)) = \langle T, S \rangle$, and $Ax(f) = \{\mathsf{A}, \mathsf{C}\}$. We say that P is (k, l)-distinct if each $h \in T \cap S$ occurs at most k times in w_1 and at most k times in w_2, the number of symbols in $T \cap S \leq l$ and $T \setminus (T \cap S) \equiv \emptyset$ iff $S \setminus (T \cap S)\emptyset$. We say $P \equiv X(\vec{x}) : \lambda x_1, \ldots, x_w.t \triangleq \lambda y_1, \ldots, y_r.s$ is *total* (k, l)-*distinct* if $|t| = |s| = 1$ or for every pair of subterms (t', s') of t and s such that $head(t') = head(s')$, the AUP $Y(\vec{y}) : t' \triangleq s'$ is total (k, l)-distinct.

This concept is much simpler than k-determined in that it basically splits the arguments of the left and right side of the given AUP into at most l sections dependent on the head symbols of the arguments. Also, for head function symbol, there should be at most k occurrences of it and the result of decomposition is an empty term iff the terms of the left and right side of the AUP are empty.

When an AUP is total $(1, l)$-distinct there is only one way to decompose the AUP, i.e. either a given symbol shows up in both w_1 and w_2 once and can be aligned, or it cannot be aligned and is generalized by a new variable. This leads to the following results:

▶ **Theorem 24.** *A higher-order* $\{\mathsf{A}, \mathsf{C}\}$-*refined pattern generalization for a* total $(1, l)$-*distinct AUP can be solved in linear time.*

Proof. Similar to the proof of Theorem 10. ◀

If we attempt to relax these constraints the time complexity of the algorithm increases substantially, even when we consider the case of $(2, l)$-distinct AUPs under our restricted optimality condition.

▶ **Definition 25.** Let $w_1 = (a_1, \ldots, a_n)$ and $w_2 = (b_1, \ldots, b_m)$ be sequences of symbols. We define the singleton rigidity function $\mathcal{R}_{AC}^{(k,l)}$ as follows

$$\mathcal{R}_{AC}^{(k,l)}(w_1, w_2) = \left\{ \begin{array}{c} \left\{ a_l\,[i, j] \;\middle|\; a_i = b_j, 1 \leq i \leq n1 \leq j \leq m \right\} \\ \emptyset \end{array} \;\middle|\; \begin{array}{c} \text{if } (w_1, w_2) \text{ is } (k, l)\text{-distinct} \\ \text{otherwise} \end{array} \right. \tag{3}$$

▶ **Theorem 26.** *A* $(\mathcal{R}_{AC}^{(k,l)}, C_{(\mathcal{R}_{AC}^{(k,l)}, \mathcal{G}_{AC})}, \mathcal{G}_{AC})$-*optimal higher-order* $\{\mathsf{A}, \mathsf{C}\}$-*refined pattern generalization for a* total (k, l)-*distinct AUP is computed in* $O(k^{2 \cdot l} \cdot n^2)$ *time where* n *is the input size.*

Proof. There are $O(k^2)$ ways to pair the terms with the same head and there are l blocks thus there are $O(k^{2 \cdot l})$ computations using $\mathcal{G}_{base}$ (complexity $O(n)$) to be performed on an AUP with size n. ◀

Obviously, computing the full set of E-refined generalizations from the results of Theorem 26 using a naive method would take in the order of $O(k^{2 \cdot l \cdot n})$ time.

9   Conclusion

The higher-order equational anti-unification algorithm presented in this paper combines higher-order syntactic anti-unification rules with the decomposition rules for associative, commutative and associative-commutative function symbols. This gives a modular algorithm, which can be used for problems with different symbols from different theories without any adaptation.

Higher order A-, C-, and AC-anti-unification problems are finitary. In practice, often it is desirable to compute only one answer, which is the best one with respect to some predefined criterion. We defined such an optimality criterion, which basically means that an optimal equational solution should be at least a good as the syntactic lgg. We then identified problem forms for which optimal solutions can be computed fast (in linear or polynomial time) by a greedy approach.

References

1   María Alpuente, Santiago Escobar, Javier Espert, and José Meseguer. A modular order-sorted equational generalization algorithm. *Inf. Comput.*, 235:98–136, 2014. `doi:10.1016/j.ic.2014.01.006`.

2   Henk Barendregt. *The Lambda Calculus. Its Syntax and Semantics*. North Holland, 1984.

3   Adam D. Barwell, Christopher Brown, and Kevin Hammond. Finding parallel functional pearls: Automatic parallel recursion scheme detection in Haskell functions via anti-unification. *Future Generation Comp. Syst.*, 79:669–686, 2018. `doi:10.1016/j.future.2017.07.024`.

4   Alexander Baumgartner. *Anti-Unification Algorithms: Design, Analysis, and Implementation*. PhD thesis, Johannes Kepler University Linz, 2015.

5   Alexander Baumgartner and Temur Kutsia. A library of anti-unification algorithms. In Eduardo Fermé and João Leite, editors, *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014. Proceedings*, volume 8761 of *Lecture Notes in Computer Science*, pages 543–557. Springer, 2014. `doi:10.1007/978-3-319-11558-0_38`.

6   Alexander Baumgartner and Temur Kutsia. Unranked second-order anti-unification. *Inf. Comput.*, 255:262–286, 2017. `doi:10.1016/j.ic.2017.01.005`.

7   Alexander Baumgartner, Temur Kutsia, Jordi Levy, and Mateu Villaret. A variant of higher-order anti-unification. In Femke van Raamsdonk, editor, *24th International Conference on Rewriting Techniques and Applications, RTA 2013*, volume 21 of *LIPIcs*, pages 113–127. Schloss Dagstuhl, 2013.

8   Alexander Baumgartner, Temur Kutsia, Jordi Levy, and Mateu Villaret. Higher-order pattern anti-unification in linear time. *J. Autom. Reasoning*, 58(2):293–310, 2017.

9   Tarek R. Besold, Kai-Uwe Kühnberger, and Enric Plaza. Towards a computational- and algorithmic-level account of concept blending using analogies and amalgams. *Connect. Sci.*, 29(4):387–413, 2017. `doi:10.1080/09540091.2017.1326463`.

10 Gilles Dowek. Higher-order unification and matching. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 1009–1062. Elsevier and MIT Press, 2001.

11 Temur Kutsia, Jordi Levy, and Mateu Villaret. Anti-unification for unranked terms and hedges. *J. Autom. Reasoning*, 52(2):155–190, 2014. `doi:10.1007/s10817-013-9285-6`.

12 Tomer Libal and Alexander Steen. Towards a substitution tree based index for higher-order resolution theorem provers. In Pascal Fontaine, Stephan Schulz, and Josef Urban, editors, *Proceedings of the 5th PAAR Workshop*, volume 1635 of *CEUR Workshop Proceedings*, pages 82–94. CEUR-WS.org, 2016. URL: `http://ceur-ws.org/Vol-1635/paper-08.pdf`.

13 Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Log. Comput.*, 1(4):497–536, 1991. `doi:10.1093/logcom/1.4.497`.

14 Frank Pfenning. Unification and anti-unification in the calculus of constructions. In *LICS*, pages 74–85. IEEE Computer Society, 1991.

15 Brigitte Pientka. Higher-order term indexing using substitution trees. *ACM TOCL*, 11(1), 2009. `doi:10.1145/1614431.1614437`.

16 Ute Schmid. *Inductive Synthesis of Functional Programs, Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning*, volume 2654 of *Lecture Notes in Computer Science*. Springer, 2003.

17 Martin Schmidt, Ulf Krumnack, Helmar Gust, and Kai-Uwe Kühnberger. Heuristic-driven theory projection: An overview. In Henri Prade and Gilles Richard, editors, *Computational Approaches to Analogical Reasoning: Current Trends*, volume 548 of *Studies in Computational Intelligence*, pages 163–194. Springer, 2014. `doi:10.1007/978-3-642-54516-0_7`.

Term Rewriting Characterisation of LOGSPACE for Finite and Infinite Data

Łukasz Czajka[1]

University of Copenhagen, Denmark
luta@di.ku.dk

─── **Abstract** ───

We show that LOGSPACE is characterised by finite orthogonal tail-recursive cons-free constructor term rewriting systems, contributing to a line of research initiated by Neil Jones. We describe a LOGSPACE algorithm which computes constructor normal forms. This algorithm is used in the proof of our main result: that simple stream term rewriting systems characterise LOGSPACE-computable stream functions as defined by Ramyaa and Leivant. This result concerns characterising logarithmic-space computation on infinite streams by means of infinitary rewriting.

1   Introduction

The goal of the field of implicit computational complexity is to characterise computational complexity classes without reference to external measuring conditions. One of the first such implicit characterisations was that of LOGSPACE as the class of problems which can be decided by deterministic two-way multihead finite automata [6]. Inspired by this well-known characterisation, Neil Jones gave new characterisations of this class as "cons-free" tail-recursive programs in several formalisms [9, 7, 8]. In cons-free programs data constructors cannot occur in function bodies. Put differently, cons-free programs are read-only: recursive data can only be read from input, but not created or altered (except taking subterms). Cons-free programming was subseqently used to characterise a variety of complexity classes [9, 7, 8, 2, 3, 10, 11, 12].

In this paper we extend the cons-free approach to computation on infinite streams. In [14, 13] Ramyaa and Leivant define the class of LOGSPACE-computable stream functions and show that it is characterised by ramified corecurrence in two tiers. Our main contribution is a cons-free infinitary term-rewriting characterisation of this class. We show that a stream function is computable in LOGSPACE, in the sense of Ramyaa and Leivant, if and only if it is definable in a simple stream TRS. As an intermediate step, we also give infinitary rewriting characterisations of stream functions computable by (jumping) finite stream transducers.

In order to obtain our characterisation of LOGSPACE-computability on streams, we give an algorithm to compute the (finite) constructor normal form of a (finite) term of a certain form in a finite orthogonal tail-recursive cons-free constructor TRS. Using this algorithm we obtain a term rewriting characterisation of LOGSPACE (in the ordinary finite sense).

───

In previous work [9, 8, 2] LOGSPACE was characterised by tail-recursive cons-free programs. The idea to transpose characterisations obtained via cons-free programs into the formalism of TRSs has already been exploited to characterise other complexity classes in [3, 11, 10], but there orthogonality was not assumed. Our method of introducing ⊥-reductions may be seen as a degenerate case of the method in [3] (see also [10]), but the algorithm used there to compute constructor normal forms in polynomial time is fundamentally different from ours and does not easily adapt to logarithmic space computation. In the first part of this paper, the main novelty is a trick to detect looping in logarithmic space, and using this to obtain a LOGSPACE algorithm for computing constructor normal forms.

2   Term rewriting systems

We assume familiarity with term rewriting [1]. In this short section we fix the notation and briefly recall some definitions.

▶ **Definition 2.1.** A *term rewriting system* (TRS) is a set of *rules* of the form $l \to r$ where l, r are terms and l is not a variable and $\mathrm{Var}(r) \subseteq \mathrm{Var}(l)$, where $\mathrm{Var}(t)$ denotes the variables occurring in t. Given a TRS R, the reduction relation $\to_R$ is the compatible closure of the contraction relation $\{(\sigma l, \sigma r) \mid l \to r \in R, \sigma$ a substitution$\}$. We use $\to^*$ for the transitive-reflexive closure of $\to$, and $\to^=$ for the reflexive closure, and $\rightrightarrows$ for the parallel closure. For precise definitions see [1]. In particular, $\rightrightarrows$ is reflexive.

A *defined symbol* in a TRS R is a function symbol which occurs at the root of a left-hand side of a rule in R. A *constructor symbol* in a TRS R is a function symbol which is not a defined symbol in R. A *constructor term* is a term which does not contain defined function symbols (it may contain variables). A *constructor normal form* is a constructor term which does not contain variables (so it contains only constructors). A *constructor head normal form* (chnf) is a term of the from $c(t_1, \ldots, t_n)$ with c a constructor. A *constructor TRS* is a TRS R such that for $l \to r \in R$ we have $l = f(l_1, \ldots, l_n)$ where $l_1, \ldots, l_n$ are constructor terms.

A redex is *innermost* if it does not contain other redexes. A reduction step is innermost if it contracts an innermost redex.

A *decision problem* is a set of binary words $A \subseteq \{0, 1\}^*$. Assuming the signature contains the constants $0, 1, \mathrm{nil}$ and a binary constructor symbol cons, every $w \in \{0, 1\}^*$ may be represented by a term $\bar{w}$ in an obvious way. A TRS R *accepts* a decision problem A if there is a function symbol f such that for every $w \in \{0, 1\}^*$ we have: $f(\bar{w}) \to_R^* 1$ iff $w \in A$.

3   LOGSPACE for finite data

In this section we show that finite orthogonal tail-recursive cons-free constructor TRSs characterise LOGSPACE, i.e., a decision problem is in LOGSPACE iff it is accepted by a finite orthogonal tail-recursive cons-free constructor TRS. As part of the proof we give an algorithm which computes the constructor normal form of a term of a certain form, if there exists one, or rejects otherwise. This algorithm will also be used in Section 6.

▶ **Definition 3.1.** A constructor TRS R is *cons-free* if for each $l \to r \in R$ every chnf subterm of r either occurs in l or is a constructor normal form. A constructor TRS R is *tail-recursive* if there is a preorder $\succsim$ on defined function symbols such that for every $f(u_1, \ldots, u_n) \to r \in R$ and every defined function symbol g the following hold:
- if $r = g(t_1, \ldots, t_k)$ then $f \succsim g$,
- if $g(t_1, \ldots, t_k)$ is a proper subterm of r then $f > g$.

A TRS is *strictly tail-recursive* if it is tail-recursive and each right-hand side of a rule contains at most one defined function symbol.

For terms $t_1, \ldots, t_n$ by $\mathcal{B}(t_1, \ldots, t_n)$ we denote the sets of all constructor normal forms occurring either in one of t_i or in a right-hand side of a rule of R. Note that $\mathcal{B}(t_1, \ldots, t_n)$ is finite if R is.

Our definition of tail-recursiveness is based on standard definitions in the literature [8, 2], adapted to the term rewriting framework.

▶ **Proposition 3.2.** *Any problem decidable in LOGSPACE is accepted by a finite orthogonal tail-recursive cons-free constructor TRS.*

Proof. This is a straightforward adaptation of previous work [7, 2]. One may e.g. easily encode any $CM^{\backslash+}$ program from [7] by a finite orthogonal strictly tail-recursive cons-free constructor TRS. Because the obtained TRS is orthogonal and strictly tail-recursive, the reduction strategy does not play a significant role. We skip the routine details. ◀

It is more difficult to show the other direction of the characterisation result, i.e., that any decision problem accepted by a finite orthogonal tail-recursive cons-free constructor TRS is in LOGSPACE. Indeed, if the TRS is tail-recursive but not strictly tail-recursive, then terms which have a constructor normal form may also have arbitrarily large reducts. Consider e.g. the following TRS R:

$$f(x) \to_R f(g(x)) \qquad h(x) \to_R a$$

Then $h(f(a)) \to_R a$ but also $h(f(a)) \to_R^* h(f(g^n(a)))$ for any $n \in \mathbb{N}$. This example also shows that the innermost strategy may fail to give a normal form even if a term has one.

We will show that a constructor normal form may always be reached by an eager $R\perp$-reduction, denoted $\to_{R\perp e}^*$, which contracts only innermost R-redexes and eagerly (as soon as possible) replaces by $\perp$ an innermost subterm with no constructor normal form in R. For instance, in the example TRS R given above $h(f(a)) \to_\perp h(\perp) \to_R a$ is an eager $R\perp$-reduction, but $h(f(a)) \to_R h(f^2(a))$ is not. The term $f(a)$ does not have a constructor normal form in R, so it *cannot* be R-contracted in an eager $R\perp$-reduction – it *must* be contracted to $\perp$.

Whether a subterm has a constructor normal form in R may be decided using a constant number of logarithmic counters. An eager $R\perp$-reduction has the form

$$f_1(w_1^1, \ldots, w_{n_1}^1) \to_{R\perp e}^* f_1(t_1^1, \ldots, t_{n_1}^1) \to_R^\epsilon f_2(w_1^2, \ldots, w_{n_2}^2) \to_{R\perp e}^* f_2(t_1^2, \ldots, t_{n_2}^2) \to_R^\epsilon \cdots$$

where t_i^j is the constructor normal form w.r.t. eager $R\perp$-reduction of w_i^j ($\perp$ is considered to be a constructor) and $f_i \gtrsim f_j$ for $i \leq j$. At some point either we reach a constructor normal form or a term $f_i(t_1^i, \ldots, t_{n_i}^i)$ repeats. Because of cons-freeness, there are only polynomially many such terms. Hence, a logarithmic counter may be used to detect looping. Because of tail-recursiveness, computing the constructor normal form (w.r.t. eager $R\perp$-reduction) t_i^j of w_i^j may be done by a recursive invocation, and the recursion depth will be constant. The rest of this section is devoted to making the above arguments precise.

▶ **Definition 3.3.** Let R be a constructor TRS and let $\perp$ be a fresh constant, i.e., not occurring in any of the rules of R. We define the $\perp$-*contraction* relation $\to_\perp^\epsilon$ by: $t \to_\perp^\epsilon \perp$ if t does not R-reduce to a constructor normal form. The $\perp$-*reduction* relation $\to_\perp$ is the compatible closure of $\to_\perp^\epsilon$. We set $\to_{R\perp} = \to_R \cup \to_\perp$. An $R\perp$-reduction is *eager* if only innermost $R\perp$-redexes are contracted and priority is given to $\perp$-reduction, i.e., an R-redex t such that $t \to_\perp \perp$ is not R-contracted in the reduction. We use $\to_{R\perp e}$ for an eager one-step $R\perp$-reduction.

Note that $\perp$ is a constructor. So a term of the form $c(t_1, \ldots, t_n)$ with c a constructor never eagerly $R\perp$-reduces to $\perp$, because if it does not have a constructor normal form in R then there is a $R\perp$-redex in one of the t_i. Note that a term is in normal form w.r.t. $R\perp$-reduction iff it is a constructor normal form.

We first show that in a left-linear constructor TRS $\perp$-reduction may be postponed after R-reduction. This will imply that eager $R\perp$-reduction to a constructor normal form not containing $\perp$ may be replaced with R-reduction.

▶ **Lemma 3.4.** *In a left-linear constructor TRS, if $u \rightrightarrows_\perp t \rightarrow_R t'$ then there is u' with $u \rightarrow_R u' \rightrightarrows_\perp t'$.*

Proof. Without loss of generality we may assume that $t \rightarrow_R t'$ occurs at the root by a rule $l \rightarrow r$ with substitution σ. By the choice of $\perp$ the term l does not contain $\perp$. We have $t = \sigma(l)$. So $\perp$ in t may occur only below a variable position of l. Since $\perp$ are the contracta in $u \rightrightarrows_\perp t$, the expansions $u \rightrightarrows_\perp t$ in t occur below variable positions of l. Hence, there is σ' such that $\sigma'(x) \rightrightarrows_\perp \sigma(x)$ for all $x \in \text{Var}(l)$ and $u = \sigma'(l)$. Then take $u' = \sigma'(r)$. ◀

▶ **Corollary 3.5.** *In a left-linear constructor TRS, if $t \rightarrow^*_{R\perp} t'$ then there is u with $t \rightarrow^*_R u \rightarrow^*_\perp t'$.*

▶ **Lemma 3.6.** *In a left-linear constructor TRS, if $t \rightarrow^*_{R\perp} s$ with s a constructor normal form not containing $\perp$, then $t \rightarrow^*_R s$.*

Proof. Induction on the number n of $\perp$-contractions in $t \rightarrow^*_{R\perp} s$. If $n > 0$ then consider the last $\perp$-contraction: $t \rightarrow^*_{R\perp} t' \rightarrow_\perp t'' \rightarrow^*_R s$. By Lemma 3.4 there is s' with $t' \rightarrow^*_R s' \rightrightarrows_\perp s$. Because s does not contain $\perp$, we have $s' = s$. So $t \rightarrow^*_{R\perp} s$ with $n - 1$ $\perp$-contractions. Hence $t \rightarrow^*_R s$ by the inductive hypothesis. ◀

The following lemma shows that eager $R\perp$-reduction in $\sigma(t)$, with t a linear constructor term, occurs below variable positions.

▶ **Lemma 3.7.** *In a constructor TRS R, if t is a linear constructor term and $\sigma(t) \rightarrow^*_{R\perp e} t'$ then there is σ' such that $t' = \sigma'(t)$ and $\sigma(x) \rightarrow^*_{R\perp e} \sigma'(x)$ for all $x \in \text{Var}(t)$.*

Proof. Induction on t. If $t = x$ then take $\sigma'(x) = t'$. Otherwise $t = c(t_1, \ldots, t_n)$ and $t' = c(t'_1, \ldots, t'_n)$ with $\sigma(t_i) \rightarrow^*_{R\perp e} t'_i$ and c a constructor. By the inductive hypothesis for $i = 1, \ldots, n$ there is σ'_i with $\sigma'_i(t_i) = t'_i$ and $\sigma(x) \rightarrow^*_{R\perp e} \sigma'_i(x)$ for $x \in \text{Var}(t_i)$. Because t is linear, $\text{Var}(t_i) \cap \text{Var}(t_j) = \emptyset$ for $i \neq j$. So the σ'_is may be combined into a single substitution σ' with the required properties. ◀

▶ **Corollary 3.8.** *In a left-linear constructor TRS R, if $f(t_1, \ldots, t_n) \rightarrow^\epsilon_R t$ and $t_i \rightarrow^*_{R\perp e} t'_i$ for $i = 1, \ldots, n$, then there is t' with $f(t'_1, \ldots, t'_n) \rightarrow^\epsilon_R t' {}^*_{R\perp e}\!\leftarrow t$. Moreover, the contraction $f(t'_1, \ldots, t'_n) \rightarrow^\epsilon_R t'$ is by the same rule as $f(t_1, \ldots, t_n) \rightarrow^\epsilon_R t$.*

Proof. Assume $f(t_1, \ldots, t_n) \rightarrow^\epsilon_R t$ by a rule $f(l_1, \ldots, l_n) \rightarrow r$ with substitution σ. Because each l_i is a linear constructor term and $\text{Var}(l_i) \cap \text{Var}(l_j) = \emptyset$ for $i \neq j$, by Lemma 3.7 there is σ' such that for $i = 1, \ldots, n$ we have $\sigma'(l_i) = t'_i$ and $\sigma(x) \rightarrow^*_{R\perp e} \sigma'(x)$. Thus $u = f(\sigma'(l_1), \ldots, \sigma'(l_n)) \rightarrow_R \sigma'(r)$. Also $t' = \sigma(r) \rightarrow^*_{R\perp e} \sigma'(r)$, because $\text{Var}(r) \subseteq \text{Var}(l_1, \ldots, l_n)$. So we may take $t' = \sigma'(r)$. ◀

The next lemma shows a strengthening of the diamond property for eager $R\perp$-reduction in orthogonal TRSs.

▶ **Lemma 3.9.** *In an orthogonal TRS R, if $t \to_{R \perp e} t_1$ and $t \to_{R \perp e} t_2$ then either $t_1 = t_2$ or there is t' with $t_1 \to_{R \perp e} t'$ and $t_2 \to_{R \perp e} t'$.*

Proof. If the redexes are parallel then the second part of the disjunction holds. Because both redexes are innermost, if they are not parallel we may assume without loss of generality that both of them are at the root. If both of them are R-redexes, then $t_1 = t_2$ by orthogonality. If both are $\perp$-redexes then $t_1 = t_2 = \perp$. It is not possible that one redex is a $\perp$-redex and the other an R-redex, because the reductions are eager. ◀

The following simple lemma is needed in the proof of Lemma 3.11.

▶ **Lemma 3.10.** *In a cons-free constructor TRS, if every subterm of t in chnf is in constructor normal form and $t \to_R^* t'$ and t' is in chnf, then t' is in constructor normal form.*

Proof. Because the TRS is cons-free, any chnf subterm of any R-reduct of t must be in $\mathcal{B}(t)$. More precisely, one shows that if $t \to_R u$ then still every subterm of u in chnf is in constructor normal form. ◀

In the rest of this section we assume that R is a finite orthogonal tail-recursive cons-free constructor TRS.

Note that because R is finite and tail-recursive the partial order on the equivalence classes determined by $\gtrsim$ may be extended to a well order $>_E$. We write $t_1 >_E t_2$ ($t_1 \geq_E t_2$) if the greatest equivalence class of a defined function symbol in t_1 is greater (greater or equal) than the greatest equivalence class of a defined function symbol in t_2. We write $f \leq_E t$ if the equivalence class of the defined function symbol f is less or equal to the greatest equivalence class of a defined function symbol in t. Note that if $t \to_{R\perp}^* t'$ then $t \geq_E t'$, because R is tail-recursive.

Our next goal is to show that every term has a constructor normal form (possibly containing $\perp$) reachable by eager $R\perp$-reduction. This will imply that eager $R\perp$-reduction commutes with R-reduction, and that eager $R\perp$-reduction is terminating.

▶ **Lemma 3.11.** *Assume that for all t' with $t' \leq_E t$ there is s in constructor normal form such that $t' \to_{R \perp e}^* s$. If $t' \; _R \leftarrow t \to_{R \perp e} u$ then there is u' with $t' \to_{R \perp e}^* u' \; _{\overline{R}} \leftarrow u$.*

Proof. Note that because the redex contracted in $t \to_{R \perp e} u$ is innermost, it cannot happen that the redex contracted in $t \to_R t'$ occurs strictly inside this redex. So we may assume without loss of generality that the redex contracted in $t \to_R t'$ occurs at the root.

If $u = \perp$ then $t = f(s_1, \ldots, s_n)$ with $s_1, \ldots, s_n$ in constructor normal form, because the $R\perp$-reduction is innermost. Since $t' \leq_E t$, there is a constructor normal form s such that $t' \to_{R \perp e}^* s$. If $s = \perp$ then we may take $u' = \perp$. Otherwise, $s = c(s_1', \ldots, s_m')$ with $c \neq \perp$ a constructor. By Corollary 3.5 there is w with $t \to_R t' \to_R^* w \to_\perp^* s$. Then w is in chnf. But then by Lemma 3.10 it is in constructor normal form. This contradicts $t \to_\perp \perp$.

If $t \to_{R \perp e} u$ contracts an R-redex at the root, then $u = t'$ because R is orthogonal, so take $u' = t'$. The remaining case, when the eager $R\perp$-contraction occurs strictly below the root, follows from Corollary 3.8. ◀

▶ **Lemma 3.12.** *Assume that for all t with $t <_E g$ there is s in constructor normal form such that $t \to_{R \perp e}^* s$. If $g(t_1, \ldots, t_n) \to_R^* s$ with g a defined function symbol and s in constructor normal form and $t_i <_E g$ and $t_i \to_{R \perp e}^* w_i$ for $i = 1, \ldots, n$, then $g(w_1, \ldots, w_n) \to_R^* s$.*

Proof. Induction on the number of root steps in $g(t_1, \ldots, t_n) \to_R^* s$. There is at least one root step, so $g(t_1, \ldots, t_n) \to_R^* g(t_1', \ldots, t_n') \to_R^\epsilon t \to_R^* s$ and, because R is cons-free and tail-recursive, either $t = s$ or $t = g'(u_1, \ldots, u_m)$ with $g' \leq_E g$ and $u_i <_E g$. By Lemma 3.11 there are $w_1', \ldots, w_n'$ such that $w_i \to_R^* w_i'$ and $t_i' \to_{R \perp e}^* w_i'$ for $i = 1, \ldots, n$. By Corollary 3.8 there is t' with $g(w_1', \ldots, w_n') \to_R^\epsilon t'$ and $t \to_{R \perp e}^* t'$. If $t = s$ then $t' = s$ and we are done. Otherwise, by Corollary 3.8, $t' = g'(u_1', \ldots, u_m')$ and $u_i \to_{R \perp e}^* u_i'$. By the inductive hypothesis $t' \to_R^* s$. Hence $g(w_1, \ldots, w_n) \to_R^* g(w_1', \ldots, w_n') \to_R t' \to_R^* s$.   ◄

▶ **Lemma 3.13.** *Assume that for all t with $t <_E g$ there is s in constructor normal form such that $t \to_{R \perp e}^* s$. If $g(t_1, \ldots, t_n) \to_R^* s$ with g a defined function symbol and s in constructor normal form and $t_i <_E g$ for $i = 1, \ldots, n$, then $g(t_1, \ldots, t_n) \to_{R \perp e}^* s$.*

Proof. The reduction $g(t_1, \ldots, t_n) \to_R^* s$ has the form:

$$g(t_1, \ldots, t_n) \to_R^* g(u_1, \ldots, u_n) \to_R^\epsilon g_1(t_1^1, \ldots, t_{n_1}^1) \to_R^* g_1(u_1^1, \ldots, u_{n_1}^1) \to_R^\epsilon \ldots \to_R^\epsilon s.$$

We proceed by induction on the number of root steps in this R-reduction. Since $t_i <_E g$ for $i = 1, \ldots, n$, there are $s_1, \ldots, s_n$ in constructor normal form such that $t_i \to_{R \perp e}^* s_i$. By Lemma 3.11 we also have $u_i \to_{R \perp e}^* s_i$. By Corollary 3.8 there is t' with $g(s_1, \ldots, s_n) \to_R^\epsilon t'$ and either $t' = s$, or $t' = g_1(w_1, \ldots, w_{n_1})$ and $t_i^1 \to_{R \perp e}^* w_i$. We have $g(s_1, \ldots, s_n) \to_{R \perp e} s$ because the R-reduction to t' is innermost and $g(s_1, \ldots, s_n) \to_R^* s$ by Lemma 3.12. Hence if $t' = s$ then $g(t_1, \ldots, t_n) \to_{R \perp e}^* s$. So assume $t' = g_1(w_1, \ldots, w_{n_1})$ with $t_i^1 \to_{R \perp e}^* w_i$. By the inductive hypothesis $g_1(t_1^1, \ldots, t_{n_1}^1) \to_{R \perp e}^* s$. By Lemma 3.9 we obtain $g_1(w_1, \ldots, w_{n_1}) \to_{R \perp e}^* s$. Thus $g(t_1, \ldots, t_n) \to_{R \perp e}^* g(s_1, \ldots, s_n) \to_{R \perp e} g_1(w_1, \ldots, w_{n_1}) \to_{R \perp e}^* s$.   ◄

▶ **Lemma 3.14.** *For every term t there exists s in constructor normal form[2] such that $t \to_{R \perp e}^* s$.*

Proof. We proceed by induction on pairs $\langle e, n \rangle$ ordered lexicographically, where e is the greatest, w.r.t. $>_E$, equivalence class of a defined function symbol in t, and n is the size of t. This is obvious if t is a variable. So assume $t = f(t_1, \ldots, t_n)$. Since each t_k is smaller than t, by the inductive hypothesis for each $k = 1, \ldots, n$ there is a constructor normal form s_k with $t_k \to_{R \perp e}^* s_k$. If f is a constructor then we are done, so assume it is a defined function symbol. If $f(s_1, \ldots, s_n)$ does not R-reduce to a constructor normal form, then $f(s_1, \ldots, s_n) \to_{R \perp e} \perp$, so we may take $s = \perp$. Otherwise $f(s_1, \ldots, s_n) \to_R^* s$ for some s in constructor normal form. Of course, $f \leq_E t$, so the inductive hypothesis implies that for all t' with $t' <_E f$ there is s' in constructor normal form such that $t' \to_{R \perp e}^* s'$. Thus by Lemma 3.13: $t \to_{R \perp e}^* f(s_1, \ldots, s_n) \to_{R \perp e}^* s$.   ◄

▶ **Corollary 3.15.** *If $t' \,{}_R^* \!\leftarrow t \to_{R \perp e}^* u$ then there is u' with $t' \to_{R \perp e}^* u' \,{}_R^* \!\leftarrow u$.*

Proof. Follows from Lemma 3.11 and Lemma 3.14.   ◄

▶ **Remark.** Corollary 3.15 fails if the $R \perp$-reduction is not required to be eager (though innermost would suffice). Consider the TRS R:

$$f(x) \to_R f(x) \qquad g(c(x)) \to_R a$$

We have $g(c(f(x))) \to_R a$, but also $g(c(f(x))) \to_\perp g(\perp) \to_\perp \perp$, because $c(f(x))$ does not R-reduce to a constructor normal form.

---

[2] Recall that $\perp$ is considered to be a constructor.

The corollary also fails if R is not required to be cons-free. Consider the TRS R:

$$f(x) \to_R f(x) \qquad g(x) \to_R c(f(x))$$

Then $g(x) \to_{R\perp e}^* \perp$. On the other hand $g(x) \to_R c(f(x))$ and $c(f(x)) \not\to_{R\perp e}^* \perp$.

If R is not required to be tail-recursive then this also fails. Consider the TRS R:

$$h(x) \to_R h(f(x)) \qquad f(x) \to_R g(x, f(x)) \qquad g(x, y) \to_R x$$

Then $h(a) \to_{R\perp e} \perp$, because $h(t)$ does not have a constructor normal form for any t. Also $h(a) \to_R h(f(a))$. The term $h(f(a))$ has no constructor normal form, but $h(f(a)) \not\to_{R\perp e} \perp$ because the $\perp$-redex is not innermost. And there is no constructor normal form s with $f(a) \to_{R\perp e}^* s$ (note that $f(a) \to_R g(a, f(a)) \to_R a$ but the reduction is not innermost). Hence, there is no eager $R\perp$-reduction from $h(f(a))$ to $\perp$.

The proof of the next lemma is an adaptation of the standard argument that in an orthogonal TRS if a term is weakly innermost normalising then it is innermost terminating.

▶ **Lemma 3.16.** *Eager $R\perp$-reduction is terminating.*

Proof. Follows from Lemma 3.14 and Lemma 3.9. Assume there is an infinite eager $R\perp$-reduction $t_0 \to_{R\perp e} t_1 \to_{R\perp e} t_2 \to_{R\perp e} \ldots$. By Lemma 3.14 there is u in constructor normal form with $t_0 \to_{R\perp e}^* u$. Using Lemma 3.9 one shows by induction on the length of $t_0 \to_{R\perp e}^* u$ that there is an infinite eager $R\perp$-reduction starting at u. This contradicts that u is a constructor normal form. ◀

Termination of eager $R\perp$-reduction is crucial in justifying the correctness of the algorithm described in the proof of the following theorem.

▶ **Proposition 3.17.** *Let R be a finite orthogonal tail-recursive cons-free constructor TRS. There is a LOGSPACE algorithm which given a term $t = f(t_1, \ldots, t_n)$, with $t_1, \ldots, t_n$ in constructor normal form (possibly containing $\perp$), computes the constructor normal form $s \in \mathcal{B}(t, \perp)$ such that $t \to_{R\perp e}^* s$.*

Proof. Note that because R is cons-free, if $t \to_{R\perp}^* t'$ then any subterm of t' with a constructor symbol at the root is in $\mathcal{B}(t, \perp)$. Because the size of $\mathcal{B}(t, \perp)$ is polynomial (there is only a constant number of constructor normal forms occurring in right-hand sides of rules in R), constructor normal forms occurring in $R\perp$-reducts of t may be represented using a logarithmic number of bits.

Because R is a tail-recursive constructor TRS, $f(t_1, \ldots, t_n)$ either is R-irreducible, in which case it may be contracted to $\perp$, or it R-contracts (eagerly) to a constructor normal form, or it R-contracts (not necessarily eagerly) to a term $f'(t_1', \ldots, t_m')$ where f' is a defined function symbol and $f \gtrsim f'$ and for each defined function symbol g in one of $t_1', \ldots, t_m'$ we have $f > g$. Apply the procedure recursively, in depth-first order, to subterms of $t_1', \ldots, t_m'$ of the form $g(u_1, \ldots, u_k)$ with g a defined function symbol and $u_1, \ldots, u_k$ in constructor normal form. This results in $s_1, \ldots, s_m$ in constructor normal form such that $t_k' \to_{R\perp e}^* s_k$. Note that the number of defined function symbols in $t_1', \ldots, t_m'$ is constant and depends only on the rule of R applied to t. Hence only logarithmic space is needed to store (representations of) intermediate results. Note also that $f > g$ for g a defined symbol in $t_1', \ldots, t_m'$, which guarantees termination of the recursion.

So $f(t_1, \ldots, t_n) \to_R^\epsilon f'(t_1', \ldots, t_m') \to_{R\perp e}^* f'(s_1, \ldots, s_m)$ with $s_1, \ldots, s_m$ again in constructor normal form. We keep repeating the steps described in the previous paragraph,

starting with $f'(s_1, \ldots, s_m)$ now, until we reach a constructor normal form or we detect looping in which case $\perp$ is returned. Looping detection may be realised using a single counter with a logarithmic number of bits. Indeed, by repeating the steps described in the previous paragraph we obtain a reduction of the form

$$t \to_R^\epsilon f_1(w_1^1, \ldots, w_{n_1}^1) \to_{R\perp e}^* f_1(t_1^1, \ldots, t_{n_1}^1) \to_R^\epsilon f_2(w_1^2, \ldots, w_{n_2}^2) \to_{R\perp e}^* f_2(t_1^2, \ldots, t_{n_2}^2) \to_R^\epsilon \ldots$$

where the $R\perp e$-reductions occur strictly below the root. Let M be the maximum arity of a defined function symbol in R, and K the number of defined function symbols in R, and N the size of $\mathcal{B}(t)$ (note that N is bounded by the size of t plus a constant). There are at most N different constructor normal forms occurring in the $R\perp$-reducts of t, so if the above reduction contains more than KN^M root steps, then one of the root R-redexes $f_i(t_1^i, \ldots, t_{n_i}^i)$ must repeat. So we keep a counter and return $\perp$ after performing KN^M root steps if we do not stop with a constructor normal form earlier. To see that this is correct, note that if a root redex repeats then an infinite reduction of the above form may be constructed. Assume $t \to_R^* s$ for a constructor normal form s. Then the initial R-contraction $t \to_R^\epsilon f_1(w_1^1, \ldots, w_{n_1}^1)$ is eager, so $t \to_{R\perp e}^* f_1(t_1^1, \ldots, t_{n_1}^1)$, and thus $f_1(t_1^1, \ldots, t_{n_1}^1) \to_R^* s$ by Corollary 3.15. By induction on k we show that $f_k(t_1^k, \ldots, t_{n_k}^k) \to_R^* s$ and each of the root R-contractions $f_k(t_1^k, \ldots, t_{n_k}^k) \to_R^\epsilon f_{k+1}(w_1^{k+1}, \ldots, w_{n_{k+1}}^{k+1})$ is eager, i.e.

$$t \to_{R\perp e}^+ f_1(t_1^1, \ldots, t_{n_1}^1) \to_{R\perp e}^+ f_2(t_1^2, \ldots, t_{n_2}^2) \to_{R\perp e}^+ f_3(t_1^3, \ldots, t_{n_3}^3) \to_{R\perp e}^+ \cdots$$

Hence, there exists an infinite eager $R\perp$-reduction from t, which contradicts Lemma 3.16. Thus, if a root redex repeats then $t \to_\perp \perp$. So returning $\perp$ is correct in this case.

The above algorithm terminates and the recursion depth (the maximum nesting of recursive calls) is constant, because in the recursive calls for subterms of $t_1', \ldots, t_m'$ the defined function symbol at the root is strictly smaller in the preorder $\gtrsim$. Also note that in each recursive call on a subterm $g(u_1, \ldots, u_n)$ of one of $t_1', \ldots, t_m'$ the constructor normal forms $u_1, \ldots, u_n$ are in $\mathcal{B}(t, \perp)$, because then $g(u_1, \ldots, u_n)$ is a subterm of an $R\perp$-reduct of t. So $u_1, \ldots, u_n$ may still be represented in logarithmic space. Hence, at each recursive invocation the algorithm uses logarithmic space to store the representations of the function symbol arguments, a constant number of logarithmic-space variables to store the intermediate results of recursive calls, and a logarithmic counter to detect looping. Since the recursion depth is constant, the algorithm altogether uses logarithmic space. ◀

▶ **Theorem 3.18.** *A decision problem is in LOGSPACE iff it is accepted by a finite orthogonal tail-recursive cons-free constructor TRS.*

Proof. The direction from left to right follows from Proposition 3.2. For the other direction it suffices to show an algorithm which given a finite orthogonal tail-recursive cons-free constructor TRS R and a term $t = f(t_1, \ldots, t_n)$ with $t_1, \ldots, t_n$ in constructor normal form not containing $\perp$, computes in LOGSPACE the constructor normal for of t, if it has one, or rejects otherwise. The algorithm is to run the procedure from Proposition 3.17 to find a constructor normal form s with $t \to_{R\perp e}^* s$. If s does not contain $\perp$ then $t \to_R^* s$ by Lemma 3.6. Otherwise, t does not have a constructor normal form in R and we reject. Indeed, if $t \to_R^* s'$ with s' in constructor normal form then s' does not contain $\perp$ because t does not. But $s = s'$ by Corollary 3.15. ◀

4   Stream Term Rewriting Systems

In this section we define stream TRSs which allow possibly infinite stream terms. We define infinitary reduction in a stream TRS which captures the notion of a "limit" of an infinite reduction sequence.

▶ **Definition 4.1.** A *stream TRS* is a two-sorted constructor TRS with sorts s (the sort of streams) and d (the sort of finite data), finitely many defined function symbols, finitely many data constructors $c_i : d^n \to d$, and one binary stream constructor cons $: d \times s \to s$. Terms of sort s are *stream terms*. Terms of sort d are *data terms*. For stream TRSs we allow terms to be infinite. We write $t_1 :: t_2$ instead of cons $t_1 \, t_2$. If $l \to r \in R$ is a rule, then we require that l and r have the same sort.

Stream rules are the rules $l \to r$ such that l is a stream term. *Data rules* are the rules $l \to r$ such that l is a data term. A *stream (resp. data) function symbol* is a defined function symbol of type $\tau_1 \times \ldots \times \tau_n \to s$ (resp. $\tau_1 \times \ldots \times \tau_n \to d$).

A *simple stream rule* has the form:

$$f(u_1, \ldots, u_n) \to t_1 :: \ldots :: t_k :: g(w_1, \ldots, w_m)$$

where $k \geq 0$ and we require:
1. $u_1, \ldots, u_n$ are constructor terms,
2. every stream subterm of one of $t_1, \ldots, t_k, w_1, \ldots, w_m$ occurs (as a subterm) in $u_1, \ldots, u_n$,
3. if $k = 0$ then every data subterm $c(v_1, \ldots, v_j)$ of each of $w_1, \ldots, w_m$, with $c : d^j \to d$ a data constructor, either occurs in $u_1, \ldots, u_n$ or is a constructor normal form.

The intuitive interpretation of the restrictions of a simple stream rule is that it is cons-free with respect to stream subterms, and if the rule does not produce a new stream element then it is also cons-free with respect to data subterms.

Note that by requiring $u_1, \ldots, u_n$ to be constructor terms and every stream subterm of each of $t_1, \ldots, t_k, w_1, \ldots, w_m$ to occur in $u_1, \ldots, u_n$, we ensure that stream function symbols cannot occur in $t_1, \ldots, t_k, w_1, \ldots, w_m$, i.e., g is the only stream function symbol in the right-hand side. Hence, the only function symbols present in $t_1, \ldots, t_k, w_1, \ldots, w_m$ are of data sort.

▶ **Example 4.2.** Here are some examples of simple stream rules, where x, x' are stream variables, and y is a data variable, and c is a data constructor, and h is a defined data function symbol:

$$
\begin{aligned}
f(a :: x, y) &\to a :: f(x, c(y)) \\
f(a :: x, b :: x') &\to a :: b :: f(b :: x', a :: x) \\
f(a :: x) &\to a :: g(x, c(a)) \\
f(a :: x, y) &\to f(x, h(y))
\end{aligned}
$$

Here are some non-examples:

$$
\begin{aligned}
f(a :: x, y) &\to f(x, c(y)) \\
f(a :: x, b :: x') &\to a :: b :: f(g(x'), a :: x) \\
f(a :: x) &\to a :: g(b :: x, c(a)) \\
f(a :: x, h(y)) &\to f(x, h(y))
\end{aligned}
$$

▶ **Definition 4.3.** Given a stream TRS R, *infinitary R-reduction* is defined coinductively.

$$\frac{t \to_R^* t'}{t \to_R^\infty t'} \qquad \frac{t \to_R^* u :: w \quad w \to_R^\infty w'}{t \to_R^\infty u :: w'}$$

Coinductive definitions of infinitary rewriting originate from [4, 5]. Intuitively, the definition means that $t \to_R^\infty t'$ holds if this may be derived using the above rules in a possibly infinite derivation. For example, if $f(x) \to x :: f(S(x))$ is a stream rule in R, then

$f(0) \to_R^\infty 0 :: S(0) :: S(S(0)) :: \ldots$, i.e., $f(0)$ infinitarily reduces to an infinite stream of consecutive natural numbers.

The above definition differs from the standard definition of infinitary reduction via strongly convergent reduction sequences. The difference is mainly because we effectively disallow an infinitary reduction to produce an infinite nesting of defined function symbols. This eliminates the problems with confluence in infinitary rewriting. Infinitary R-reduction, defined as above, is confluent if R is finite and orthogonal. First of all, confluence holds also for finitary R-reduction.

▶ **Lemma 4.4.** *If R is finite and orthogonal then the finitary reduction relation $\to_R$ is confluent.*

Proof. Note that the terms may be infinite. But because both the left- and right-hand sides of all rules are finite, we may use virtually the same proof as in the case of ordinary orthogonal term rewriting systems, *mutatis mutandis.*   ◀

Because of space limits we delegate the proof of confluence of infinitary reduction to Appendix A. Here we only state the result.

▶ **Theorem 4.5.** *If R is finite and orthogonal then $\to_R^\infty$ is confluent, i.e., if $t \to_R^\infty t_1$ and $t \to_R^\infty t_2$ then there exists t' such that $t_1 \to_R^\infty t'$ and $t_2 \to_R^\infty t'$.*

Let Σ be an alphabet. Assuming all elements of Σ are data constants in the rewriting system, each Σ-stream (infinite word in Σ^ω) may be treated as an infinite stream term. Also, finite words over Σ may be represented as stream terms in the TRS, where after the symbols representing the word there is a term with no constructor head normal form, e.g., $a :: b :: c :: \Omega$ represents the word abc, where Ω has no chnf. Note that a stream term in chnf (Definition 2.1) has the form $u :: w$. We denote the set of terms representing finite and infinite words over Σ by $\mathcal{S}^+(\Sigma)$, and the set of terms representing infinite words by $\mathcal{S}(\Sigma)$. More precisely, the set $\mathcal{S}^+(\Sigma)$ is defined coinductively as follows.

$$\frac{t \text{ has no chnf}}{t \in \mathcal{S}^+(\Sigma)} \qquad \frac{c \in \Sigma \quad t \in \mathcal{S}^+(\Sigma)}{(c :: t) \in \mathcal{S}^+(\Sigma)}$$

For each term t in $\mathcal{S}^+(\Sigma)$ there is exactly one corresponding finite or infinite word $|t|$ in $\Sigma^{\leq\omega} = \Sigma^\omega \cup \Sigma^*$ which this term represents.

▶ **Lemma 4.6.** *Assume $t \to_R^\infty t'$. Then t has a chnf iff t' has a chnf.*

Proof. Follows from definitions and Lemma 4.4.   ◀

▶ **Corollary 4.7.** *Let R be a finite orthogonal stream TRS. If $t \to_R^\infty s$ and $t \to_R^\infty s'$ and $s, s' \in \mathcal{S}^+(\Sigma)$ then $|s| = |s'|$.*

▶ **Definition 4.8.** A *stream function* $F : (\Sigma^\omega)^n \to \Sigma^{\leq\omega}$ is *defined* by an n-ary stream function symbol f if for any $w_1, \ldots, w_n \in \Sigma^\omega$ and $s_1, \ldots, s_n \in \mathcal{S}(\Sigma)$ with $|s_i| = w_i$ we have $f(s_1, \ldots, s_n) \to_R^\infty s$ where $|s| = F(w_1, \ldots, w_n)$. A stream function is *definable* in a stream TRS if it is defined by one of its stream function symbols.

A stream TRS R is *data tail-recursive* if the data rules of R form a *single-sorted* (i.e. neither left- nor right-hand sides of data rules of R contain stream subterms) finite tail-recursive cons-free constructor TRS.

Note that if R is data tail-recursive then data terms do not contain stream subterms, because then neither data constructors nor data function symbols can have stream arguments. In particular, if $l \to t :: r$ is a rule in R, then t does not contain stream subterms.

▶ **Definition 4.9.** A *pure* stream TRS is a finite orthogonal stream TRS with simple stream rules, no data rules and no data constructors of arity > 0.

A stream TRS has *simple data* if there exists a unary data constructor $S : d \to d$ such that for every stream rule $l \to r \in R$, if t is a data subterm of r such that $\mathrm{Var}(t) \neq \emptyset$ then $t = S(t')$ or t is a variable.

A *simple* stream TRS is a finite orthogonal data tail-recursive stream TRS with simple stream rules and simple data.

▶ **Example 4.10.** Here is an example of a simple stream TRS, where x, x' are stream variables and y, y' are data variables.

$$
\begin{array}{rcl}
f(x) & \to & g(x, x, 0, 0) \\
g(y :: x, x', 0, y') & \to & y :: g(x', x', S(y'), S(y')) \\
g(0 :: x, x', S(y), y') & \to & g(x, x', y, y') \\
g(1 :: x, x', S(y), y') & \to & g(x, x', y', y')
\end{array}
$$

In this stream TRS the stream function symbol f defines a function $F : \Sigma^\omega \to \Sigma^{\leq \omega}$ such that $F(s)$ has in position n the first element of s following a block of n consecutive 0's.

The following simple stream TRS defines the Thue-Morse sequence T:

$$
\begin{array}{rclcrcl}
T & \to & f(0) & \quad & f(x) & \to & h(x, x) :: f(S(x)) \\
h(0, 0) & \to & 0 & & \tilde{h}(0, 0) & \to & 1 \\
h(0, x) & \to & h(x, x) & & \tilde{h}(0, x) & \to & \tilde{h}(x, x) \\
h(S(0), S(x)) & \to & \tilde{h}(x, x) & & \tilde{h}(S(0), S(x)) & \to & h(x, x) \\
h(S(S(x)), S(y)) & \to & h(x, y) & & \tilde{h}(S(S(x)), S(y)) & \to & \tilde{h}(x, y)
\end{array}
$$

The n-th element T_n of T is defined by the recurrence:

$$T_0 = 0 \qquad T_{2n} = T_n \qquad T_{2n+1} = 1 - T_n$$

Identifying natural numbers with their representations in the TRS, it may be shown by induction on $\langle 2m - n, n \rangle$ ordered lexicographically that the data term $h(n, m)$ reduces to T_{2m-n} and $\tilde{h}(n, m)$ to $1 - T_{2m-n}$.

5 Finite Stream Transducers

In this section we characterise the classes of stream functions computable by (jumping) finite stream transducers. In short, pure stream TRSs characterise the class of stream functions computable by jumping finite transducers, and right-linear pure stream TRSs characterise the class of stream functions computable by finite transducers. We first recall the definitions of (jumping) finite transducers from [14, 13].

▶ **Definition 5.1.** An n-ary *jumping finite transducer* (JFT) over Σ-streams with m cursors is a tuple $\langle Q, q_0, C, \gamma, \delta \rangle$ where Q is a finite set of states, q_0 is the start state, $C = \{c_1, \ldots, c_m\}$ is the set of cursors, $\gamma : C \to \{1, \ldots, n\}$ is the initial cursor configuration, and

$$\delta : Q \times \Sigma^m \to Q \times (C \to C \cup \{+\}) \times (\Sigma \cup \{\epsilon\})$$

is the transition function. Intuitively, $\delta(q, \sigma_1, \ldots, \sigma_m)$ consists of the next state, an indication of cursor movement, and an optional output symbol. A cursor may either move forward or jump to the position of another cursor. In other words, an n-ary JFT is a finite automaton with n read-only input tapes and one write-only output tape, and m cursors which can move forward on the input tapes and jump to positions of other cursors, but cannot be compared.

A *finite transducer* (FT) is a JFT such that no cursor ever jumps to the position of another (except to itself, which is equivalent to not moving). A *configuration* of a JFT consists of a state and a function $\pi : C \to \{1, \ldots, n\} \times \mathbb{N}$ which assigns to each cursor c a stream index $i \in \{1, \ldots, n\}$ and a position in the stream. The *successor configuration* K' of a configuration K is determined in the obvious way by the transition function δ. The *initial configuration* is $\langle q_0, \pi_0 \rangle$ where $\pi_0(c) = \langle \gamma(c), 0 \rangle$ for $c \in C$. A *run* of a JFT $\langle Q, q_0, C, \gamma, \delta \rangle$ is an infinite sequence of configurations $K_0, K_1, K_2, \ldots$ such that K_0 is the initial configuration and K_{n+1} is the successor configuration of K_n for each $n \in \mathbb{N}$. The function $F : (\Sigma^\omega)^n \to \Sigma^{\leq \omega}$ *computed* by a given n-ary FT (JFT) is defined in an obvious way, with $F(w_1, \ldots, w_n)$ being the output of the transducer on inputs $w_1, \ldots, w_n$. The output may be finite, because the transducer may loop.

▶ **Theorem 5.2.** *An n-ary stream function is definable in a pure stream TRS with maximum function symbol arity m iff it is computable by an n-ary JFT with m cursors.*

Proof. Let $\langle Q, q_0, C, \gamma, \delta \rangle$ be an n-ary JFT with m cursors. Without loss of generality $C = \{1, \ldots, m\}$. In the TRS we have a stream function symbol $f_q : s^m \to s$ for each state $q \in Q$. There is also the "start" stream function symbol $g : s^n \to s$. We have the rules e.g.

$$f_q(\sigma_1 :: x_1, \ldots, \sigma_m :: x_m) \to \sigma :: f_{q'}(\sigma_{\rho(1)} :: x_{\rho(1)}, x_2, \sigma_{\rho(3)} :: x_{\rho(3)}, \ldots)$$

when $\delta(q, \sigma_1, \ldots, \sigma_m) = \langle q', \rho, \sigma \rangle$ and $\rho(1), \rho(3), \ldots \in C$ and $\rho(2) = +$. Intuitively, the arguments of f_q encode the m cursors. We also have the "start" rule:

$$g(x_1, \ldots, x_n) \to f_{q_0}(x_{\gamma(1)}, \ldots, x_{\gamma(n)}).$$

Note that all of the above rules are simple stream rules and the TRS is orthogonal, so it is a pure stream TRS. It is easy to see that for each $s_1, \ldots, s_n \in \mathcal{S}(\Sigma)$ there is a bijective correspondence between the infinite runs of the JFT on $|s_1|, \ldots, |s_n|$ and infinite reductions starting at $g(s_1, \ldots, s_n)$. This implies that the function defined by g is the same as the function computed by the JFT.

For the other direction, let R be a pure stream TRS with maximum function symbol arity m, and let the n-ary symbol g define a function $F : (\Sigma^\omega)^n \to \Sigma^{\leq \omega}$ where Σ is the set of data constants in R. We construct an n-ary JFT with m cursors.

Because there are no data rules or data constructors of arity > 0, each rule is a simple stream rule of the form e.g.

$$f(a :: u :: b :: x, a :: y, c) \to d :: g(u :: b :: x, e)$$

where $a, b, c, d, e \in \Sigma$, and u is a data variable. We will encode stream function symbols by (possibly many) states. Stream arguments will correspond to cursor positions.

Let N be the maximum size of the left-hand side l of a rule $l \to r \in R$. For a function symbol f with k stream and j data arguments, and words $w_1, \ldots, w_k \in \Sigma^N$, and constants $c_1, \ldots, c_j \in \Sigma$, we add a state $q_f^{w_1, \ldots, w_k, c_1, \ldots, c_j}$. The words $w_1, \ldots, w_k$ buffer the last N symbols read from each of the cursors. Let s_i be a stream term representing the word w_i, with a variable x_i at the tail, e.g., if $w_i = abc$ then $s_i = a :: b :: c :: x_i$. Without loss of

generality assume the stream arguments of f occur before the data arguments. Because R is orthogonal, there is at most one rule $l \to r \in R$ such that l matches $f(s_1, \ldots, s_k, c_1, \ldots, c_j)$ with some substitution σ, i.e., $\sigma l = f(s_1, \ldots, s_k, c_1, \ldots, c_j)$. Note that because of the choice of N, if there is no rule $l \to r \in R$ with l matching $f(s_1, \ldots, s_k, c_1, \ldots, c_j)$, then no left-hand side of a rule unifies with $f(s_1, \ldots, s_k, c_1, \ldots, c_j)$. Assume e.g.

$$\sigma l = f(a :: b :: x, c :: d :: y, c_1)$$

and

$$\sigma r = d :: g(c :: d :: y, b :: x, d :: y, c).$$

Then in the state q_f^{ab,cd,c_1} the JFT outputs d and simultaneously sets the first cursor to the second one, the second to the first, and the third to the second. Then it reads one symbol from the second cursor and one from the third, moving them forward. Let the read symbols be a_1, a_2 respectively. The JFT then enters the state $q_g^{cd,ba_1,da_2,c}$. This behaviour may always be encoded using a finite number of states.

The JFT starts in a state q_0 with the i-th cursor initialised to the beginning of the i-th input tape, for $i = 1, \ldots, n$, and other cursors initialised arbitrarily. Then the JFT reads N symbols from each of the n input tapes, and reaches the state $q_g^{w_1, \ldots, w_n}$ where $w_i \in \Sigma^N$ is the word consisting of the symbols read from the i-th input tape.

We also add a "trash" state q_T and add appropriate transitions to q_T from other states to make δ a total function.

For any $s_1, \ldots, s_n \in \mathcal{S}(\Sigma)$ there is a bijective correspondence between the runs of the JFT on $|s_1|, \ldots, |s_n|$ and the infinite reductions starting at $g(s_1, \ldots, s_n)$, and the function computed by the JFT is the same as the function defined by g. ◀

▶ **Theorem 5.3.** *An n-ary stream function is definable in a right-linear pure stream TRS with maximum function symbol arity m iff it is computable by an n-ary FT with m cursors.*

Proof. An adaptation of the proof of Theorem 5.2. More details are in Appendix B. ◀

6 LOGSPACE for streams

In this section we show that stream functions definable in simple stream TRSs are exactly the stream functions computable in LOGSPACE as defined by Ramyaa and Leivant [14, 13]. First, we recall the definition of jumping Turing transducers from [14].

▶ **Definition 6.1.** A *jumping Turing transducer* (JTT) is defined analogously to a JFT, except that it has additional read-write work tapes with two-way cursors on them. The function computed by a JTT is defined in an obvious way. A JTT *operates in space* $f(n)$ if the computation for the first n output symbols does not involve work-tapes of length $> f(n)$. A stream function is computable in LOGSPACE if there is a JTT computing this function which operates in space $O(\log n)$.

Note that the space used by a JTT is defined in terms of the output. Time restrictions defined in terms of the output do not make much sense for JTTs, because even for FTs no restriction is placed on how long it takes to output the next symbol (e.g. consider an FT over binary streams skipping all zeros and copying all ones).

We will show that JTTs operating in LOGSPACE compute exactly the stream functions definable in simple stream TRSs. First, we generalise eager $R\bot$-reduction from Section 3 to stream TRSs.

▶ **Definition 6.2.** Let $\perp$ be a fresh nullary data constructor. We define the relation $\to_\perp$ by: $t \to_\perp \perp$ if t is a data term and it does not R-reduce to a constructor normal form. We set $\to_{R\perp} = \to_R \cup \to_\perp$. A finitary $R\perp$-reduction is *eager* if only innermost $R\perp$-redexes are contracted and priority is given to $\perp$-reduction. We denote one-step eager $R\perp$-reduction by $\to_{R\perp e}$. The relation $\to_{R\perp e}^\infty$ of *infinitary eager $R\perp$-reduction* is defined coinductively.

$$\frac{t \to_{R\perp e}^* t'}{t \to_{R\perp e}^\infty t'} \quad \frac{t \to_{R\perp e}^* u :: w \quad w \to_{R\perp e}^\infty w'}{t \to_{R\perp e}^\infty u :: w'}$$

Because of space limits, the proofs of lemmas concerning infinitary eager $R\perp$-reduction are delegated to Appendix C.

In the rest of this section we assume R to be a simple stream TRS.

▶ **Definition 6.3.** A term is *proper* if all its data subterms are finite.

If t is proper and $t \to_R t'$ then t' is also proper, because R is finite.

▶ **Lemma 6.4.** *If t is proper and $t \to_R^\infty t_1$ and $t \to_{R\perp e}^\infty t_2$ then there is t' with $t_2 \to_R^\infty t'$ and $t_1 \to_{R\perp e}^\infty t_2$.*

▶ **Lemma 6.5.** *If $s \in \mathcal{S}^+(\Sigma)$ and $s \to_{R\perp e}^\infty s'$ (resp. $s \to_R^\infty s'$), then $s \sim s'$ and $s' \in \mathcal{S}^+(\Sigma)$.*

▶ **Theorem 6.6.** *If a stream function is definable in a simple stream TRS then it is computable in LOGSPACE.*

Proof. Let $F : (\Sigma^\omega)^n \to \Sigma^{\leq\omega}$ be a function defined by an n-ary stream function symbol f_0 in a simple stream TRS R, i.e., a finite orthogonal data tail-recursive stream TRS with simple stream rules and simple data. We describe how to construct a JTT operating in LOGSPACE which computes F.

For $s_1, \ldots, s_n \in \mathcal{S}(\Sigma)$ we have $f_0(s_1, \ldots, s_n) \to_R^\infty s \in \mathcal{S}^+(\Sigma)$ where $F(|s_1|, \ldots, |s_n|) = |s|$. The constructed JTT will essentially compute an $s' \in \mathcal{S}^+(\Sigma)$ such that $f_0(s_1, \ldots, s_n) \to_{R\perp e}^\infty s'$, for a certain fixed infinitary eager $R\perp$-reduction. By Lemma 6.4 and Lemma 6.5 we then have $|s| = |s'|$.

Note that because the TRS is finite and has simple data, all constructor normal form data terms occurring in any reduction $f_0(s_1, \ldots, s_n) \to_{R\perp e}^\infty s$ have the form $S^m(t)$ where either $t \in \Sigma$ or it is one of the finitely many constructor normal form data terms occurring in the right-hand sides of the stream or data rules. Because S cannot occur in the right-hand side of a simple stream rule if no stream element is produced, and data rules are cons-free, m is at most proportional to the number of output stream elements already produced. Hence $S^m(t)$ may be represented in logspace, using a logarithmic counter for m and a constant number of bits to represent t. Because the reduction is eager and the size of right-hand sides of stream rules is bounded by a constant, using an analogon of Proposition 3.17 we obtain a JTT which computes in logarithmic space the constructor normal form of a given data term occurring in the reduction, if it has one. This JTT computes the constructor normal forms "inside-out". For a term $f(t_1, \ldots, t_k)$ first the constructor normal forms $t'_1, \ldots, t'_k$ are computed. Each t'_i has the form $S^{m_i}(u'_i)$ where u'_i is either $\perp$ or one of the finitely many constructor normal forms occurring in the right-hand sides of the rules. Then using (an analogon of) Proposition 3.17 we compute the constructor normal form of $f(t'_1, \ldots, t'_k)$. For $S(t)$ first the constructor normal form $S^m(t')$ of t is computed using Proposition 3.17, and then $S^{m+1}(t')$ is returned as the constructor normal form of t. Note that the only property the constructor normal forms needed in Proposition 3.17 is that they can be represented

using a logarithmic number of bits, and given a representation of $S(t)$ the representation of t may be computed in logarithmic space.

We construct the JTT like in Theorem 5.2, except that now the data arguments are stored in memory instead of the state. We compute constructor normal forms of data terms using Proposition 3.17. This is done eagerly, before transitioning to the state associated with the stream function symbol in the right-hand side, which ensures that the size of the "prefix" containing all defined function symbols of each data term occurring in the reduction is constant – it is bounded by the size of the right-hand side of a rule in R. More details are in Appendix C. ◀

▶ **Theorem 6.7.** *If a stream function is computable in LOGSPACE then it is definable in a simple stream TRS.*

Proof. Let $F : (\Sigma^\omega)^n \to \Sigma^{\leq \omega}$ be a function computed by a JTT operating in LOGSPACE. As shown in [14, Proposition 2.4], the function F is also computed by a JFT with a local counter, i.e., a JFT with an additional input tape which contains 1^n when computing the n-th output symbol. In other words, a 1 is appended to the local counter whenever a symbol is output by the JFT. Initially, the local counter contains the empty word. The JFT has a fixed number of cursors on the local counter, which are reset to the beginning of the local counter tape whenever a symbol is output. As with the cursors on the input, the cursors on the local counter may move to the right or jump to other cursors. Hence, they may be encoded in an analogous way as the cursors on the input stream.

A simple stream TRS defining a function computed by a JFT with a local counter may be constructed in a way analogous to the construction of a pure stream TRS in the proof of Theorem 5.2. The difference is that now every function symbol f_q corresponding to a state q has an additional data argument representing the local counter, and data arguments encoding the cursors on the local counter. The local counter contents 1^n is represented by the data term $S^n(0)$, where $S : d \to d$ and $0 : d$. If a rule associated with f_q produces a new output symbol, then in the right-hand side of the rule the local counter is "increased" by prepending S, and the data arguments encoding cursors on the local counter are set to the local counter. This may be encoded in a simple stream rule. The resulting stream TRS has simple data.

Note that the constructed simple stream TRS actually has no data rules. It is not a pure stream TRS because it has a unary data constructor S. ◀

▶ **Corollary 6.8.** *A stream function is computable in LOGSPACE iff it is definable in a simple stream TRS.*

7 Conclusions

We have shown an infinitary rewriting characterisation of LOGSPACE-computable stream functions as defined by Ramyaa and Leivant. In the realm of finite data, we proved that finite orthogonal tail-recursive cons-free constructor TRSs characterise LOGSPACE.

Our proof could probably be adapted to show that finite semi-linear [10] tail-recursive cons-free constructor TRSs characterise NLOGSPACE. In the nondeterministic case the trick with logarithmic counters is not necessary as the procedure may simply guess when to contract a subterm to $\bot$. Semi-linearity ensures that subterms containing redexes cannot get duplicated, which is crucial to show that a constructor normal form may always be reached via an eager $R\bot$-reduction.

──── **References** ────

1 F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1999.

2 G. Bonfante. Some programming languages for LOGSPACE and PTIME. In *AMAST 2006*, pages 66–80, 2006.

3 D. de Carvalho and J. G. Simonsen. An implicit characterization of the polynomial-time decidable sets by cons-free rewriting. In *RTA-TLCA 2014*, pages 179–193, 2014.

4 Jörg Endrullis, Helle Hvid Hansen, Dimitri Hendriks, Andrew Polonsky, and Alexandra Silva. A coinductive framework for infinitary rewriting and equational reasoning. In *RTA 2015*, 2015.

5 Jörg Endrullis and Andrew Polonsky. Infinitary rewriting coinductively. In *TYPES 2011*, pages 16–27, 2011.

6 Juris Hartmanis. On non-determinancy in simple computing devices. *Acta Informatica*, 1(4):336–344, Dec 1972.

7 N. D. Jones. LOGSPACE and PTIME characterized by programming languages. *Theor. Comput. Sci.*, 228(1-2):151–174, 1999.

8 N. D. Jones. The expressive power of higher-order types or, life without CONS. *J. Funct. Program.*, 11(1):5–94, 2001.

9 Neil D. Jones. *Computability and complexity - from a programming perspective*. Foundations of computing series. MIT Press, 1997.

10 C. Kop. On first-order cons-free term rewriting and PTIME. In *DICE 2016*, 2016.

11 C. Kop and J. G. Simonsen. Complexity hierarchies and higher-order cons-free rewriting. In *FSCD 2016*, pages 23:1–23:18, 2016.

12 C. Kop and J. G. Simonsen. The power of non-determinism in higher-order implicit complexity. In *ESOP 2017*, pages 668–695, 2017.

13 D. Leivant and R. Ramyaa. The computational contents of ramified corecurrence. In *FoSSaCS 2015*, pages 422–435, 2015.

14 R. Ramyaa and D. Leivant. Ramified corecurrence and logspace. *Electr. Notes Theor. Comput. Sci.*, 276:247–261, 2011.

A   Confluence of infinitary reduction

▶ **Lemma A.1.** *If $t \to_R^\infty t' \to_R^* t''$ then $t \to_R^\infty t''$.*

Proof. By coinduction. If $t \to_R^* t'$ then this is obvious. Otherwise $t' = u :: w'$ and $t \to_R^* u :: w$ and $w \to_R^\infty w'$ and $t'' = u'' :: w''$ and $u \to_R^* u''$ and $w' \to_R^* w''$. Then $t \to_R^* u'' :: w$. By coinduction also $w \to_R^\infty w''$. Hence $t \to_R^\infty u'' :: w'' = t''$. ◀

▶ **Lemma A.2.** *If $t \to_R^\infty t' \to_R^\infty t''$ then $t \to_R^\infty t''$.*

Proof. By coinduction, using Lemma A.1. ◀

▶ **Lemma A.3.** *Let R be finite and orthogonal. If $t \to_R^\infty t'$ and $t \to_R^* s$ then there is s' with $s \to_R^\infty s'$ and $t' \to_R^\infty s'$.*

Proof. By coinduction, analysing $t \to_R^\infty t'$. If $t \to_R^* t'$ then this follows from Lemma 4.4. Otherwise $t' = u :: w'$, and $t \to_R^* u :: w$ and $w \to_R^\infty w'$. By Lemma 4.4 there are u_1, w_1 such that $s \to_R^* u_1 :: w_1$ and $u \to_R^* u_1$ and $w \to_R^* w_1$. By coinduction we obtain w_2 with $w_1 \to_R^\infty w_2$ and $w' \to_R^\infty w_2$. Hence $s \to_R^\infty u_1 :: w_2$, because $s \to_R^* u_1 :: w_1$ and $w_1 \to_R^\infty w_2$; and $t' \to_R^\infty u_1 :: w_2$, because $t' = u :: w' \to_R^* u_1 :: w'$ and $w' \to_R^\infty w_2$. ◀

$$t \xrightarrow{\quad * \quad} u_1 :: w_1 \qquad u_1 \qquad\qquad w_1 \xrightarrow{\quad \infty \quad} w_2'$$

Figure 1 Proof of confluence of infinitary reduction.
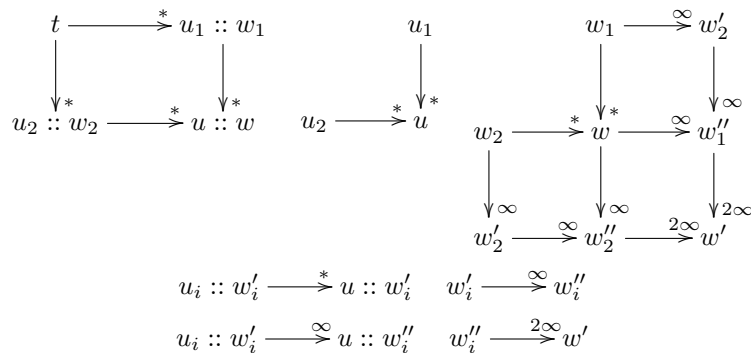
Note that $t' \to_R^* s'$ would not suffice in the conclusion of the above lemma, because the infinitary reduction $t \to_R^\infty t'$ may create in t' infinitely many descendants of a redex in t.

The relation $\to_R^{2\infty}$ is defined coinductively.

$$\frac{t \to_R^\infty t'}{t \to_R^{2\infty} t'} \qquad \frac{t \to_R^\infty u :: w \quad w \to_R^{2\infty} w'}{t \to_R^{2\infty} u :: w'}$$

▶ **Lemma A.4.** *If $t \to_R^\infty t' \to_R^{2\infty} t''$ then $t \to_R^{2\infty} t''$.*

Proof. Follows directly from Lemma A.2 ◀

▶ **Lemma A.5.** *If $t \to_R^{2\infty} t'$ then $t \to_R^\infty t'$.*

Proof. By coinduction, using Lemma A.4. ◀

▶ **Theorem 4.5.** *If R is finite and orthogonal then $\to_R^\infty$ is confluent, i.e., if $t \to_R^\infty t_1$ and $t \to_R^\infty t_2$ then there exists t' such that $t_1 \to_R^\infty t'$ and $t_2 \to_R^\infty t'$.*

Proof. By coinduction we construct t' such that $t_1 \to_R^{2\infty} t'$ and $t_2 \to_R^{2\infty} t'$. This suffices by Lemma A.5. If $t \to_R^* t_1$ or $t \to_R^* t_2$ then the claim follows from Lemma A.3. Otherwise, $t_i = u_i :: w_i'$ and $t \to_R^* u_i :: w_i$ and $w_i \to_R^\infty w_i'$ for $i = 1, 2$. By Lemma 4.4 there are u, w such that $u_i \to_R^* u$ and $w_i \to_R^* w$. By Lemma A.3 there are w_1'', w_2'' such that $w_i' \to_R^\infty w_i''$ and $w \to_R^\infty w_i''$. Hence $t_i = u_i :: w_i' \to_R^\infty u :: w_i''$. By coinduction we obtain w' with $w_i'' \to_R^{2\infty} w'$. Thus $t_i \to_R^{2\infty} u :: w'$, so we may take $t' = u :: w'$. See Figure 1. ◀

B   Characterisation of Finite Stream Transducers

▶ **Theorem 5.3.** *An n-ary stream function is definable in a right-linear pure stream TRS with maximum function symbol arity m iff it is computable by an n-ary FT with m cursors.*

Proof. First note that for an FT the construction of a stream TRS in the proof of Theorem 5.2 gives a right-linear system. Conversely, if the TRS is right-linear, then we may modify the construction of a JFT in the proof of Theorem 5.2 to obtain an FT, by keeping in the state the information which cursor a given function argument corresponds to. So a state corresponding to a function symbol f is now $q_f^{w_1,\ldots,w_k,c_1,\ldots,c_j,i_1,\ldots,i_k}$ where $i_1,\ldots,i_k$ indicate the cursors corresponding to the stream arguments of f. For instance, if

$$\sigma l = f(a :: b :: x, c :: d :: y, c_1)$$

and

$$\sigma r = d :: h(c :: d :: y, b :: x, c)$$

then the transition from the state q_f^{ab,cd,c_1,i_1,i_2} is constructed as follows. First, output d and read one symbol e from the i_1-th cursor moving it forward. Then change the state to q_h^{cd,be,c,i_2,i_1}. ◀

C Proofs for Section 6

In this section we assume that R is a simple stream TRS.

▶ **Lemma C.1.** *If t is proper and $t \to_R^\infty t_1$ and $t \to_{R\perp e}^* t_2$ then there is t' with $t_2 \to_R^\infty t'$ and $t_1 \to_{R\perp e}^\infty t_2$.*

Proof. By coinduction, analysing $t \to_R^\infty t_1$. If $t \to_R^* t_1$ then this follows from Corollary 3.15. Otherwise $t \to_R^* u :: w$ and $w \to_R^\infty w'$ and $t_1 = u :: w'$. By Corollary 3.15 there are u_2, w_2 with $t_2 \to_R^* u_2 :: w_2$ and $u \to_{R\perp e}^* u_2$ and $w \to_{R\perp e}^* w_2$. Note that w is proper. By coinduction we obtain w_2' with $w_2 \to_R^\infty w_2'$ and $w' \to_{R\perp e}^\infty w_2'$. Take $t' = u_2 :: w_2'$. We have $t_2 \to_R^* u_2 :: w_2$ and $w_2 \to_R^\infty w_2'$, so $t_2 \to_R^\infty t'$. Also $t_1 = u :: w' \to_{R\perp e}^* u_2 :: w'$ and $w' \to_{R\perp e}^\infty w_2'$, so $t_1 \to_{R\perp e}^\infty t'$. ◀

▶ **Lemma 6.4.** *If t is proper and $t \to_R^\infty t_1$ and $t \to_{R\perp e}^\infty t_2$ then there is t' with $t_2 \to_R^\infty t'$ and $t_1 \to_{R\perp e}^\infty t_2$.*

Proof. By coinduction, analysing $t \to_{R\perp e}^\infty t_2$. If $t \to_{R\perp e}^* t_2$ then this is a consequence of Lemma C.1. Otherwise $t \to_{R\perp e}^* u :: w$ and $w \to_{R\perp e}^\infty w'$ and $t_2 = u :: w'$. By Lemma C.1 there are u_1, w_1 such that $t_1 \to_{R\perp e}^* u_1 :: w_1$ and $u \to_R^* u_1$ and $w \to_R^\infty w_1$. Note that w is proper. By coinduction we obtain w_2 such that $w' \to_R^\infty w_2$ and $w_1 \to_{R\perp e}^\infty w_2$. Take $t' = u_1 :: w_2$. We have $t_1 \to_{R\perp e}^* u_1 :: w_1$ and $w_1 \to_{R\perp e}^\infty w_2$, so $t_1 \to_{R\perp e}^\infty t'$. Also $t_2 = u :: w' \to_R^* u_1 :: w'$ and $w' \to_R^\infty w_2$, so $t_2 \to_R^\infty t'$. ◀

▶ **Lemma C.2.** *If $t \to_{R\perp}^* u :: w$ then t has a chnf (in R).*

Proof. Induction on the number of $\perp$-reduction steps in $t \to_{R\perp}^* u :: w$. If there are none then $t \to_R^* u :: w$. Otherwise by the inductive hypothesis $t \to_R^* t' \to_\perp t'' \to_R^* u' :: w'$. Because R is finite, by the same argument as in the proof of Lemma 3.4 we conclude that $t \to_R^* t' \to_R^* u'' :: w'' \to_\perp^* u' :: w'$. ◀

▶ **Lemma 6.5.** *If $s \in \mathcal{S}^+(\Sigma)$ and $s \to_{R\perp e}^\infty s'$ (resp. $s \to_R^\infty s'$), then $s \sim s'$ and $s' \in \mathcal{S}^+(\Sigma)$.*

Proof. It suffices to notice that if t is a stream term without a chnf and $t \to_{R\perp e}^\infty t'$ (resp. $t \to_R^\infty t'$) then t' does not have a chnf either. This follows from Lemma C.2 (resp. Lemma 4.6). ◀

▶ **Theorem 6.6.** *If a stream function is definable in a simple stream TRS then it is computable in LOGSPACE.*

Proof. We describe in more detail the construction of a JTT already sketched in Section 6. The constructed JTT computes the stream $c_1 :: c_2 :: c_3 :: \ldots$ where e.g.

$$f_0(s_1, \ldots, s_n) \to_R^\epsilon t_1 :: f_1(w_1^1, \ldots, w_{k_1}^1) \to_{R\perp e}^* c_1 :: f_1(u_1^1, \ldots, u_{k_1}^1) \to_R^\epsilon$$
$$c_1 :: t_2 :: t_3 :: f_2(w_1^2, \ldots, w_{k_2}^2) \to_{R\perp e}^* c_1 :: c_2 :: c_3 :: f_2(u_1^1, \ldots, u_{k_2}^1) \to_R^\epsilon \ldots$$

and none of the u_i^j contain $R\perp$-redexes. So all of the root R-reduction steps are in fact eager $R\perp$-reductions. Note that all terms appearing in this reduction are proper.

Let N be the maximum size of the left-hand side l of a rule $l \to r \in R$. For a stream function symbol f with k stream arguments, and words $w_1, \ldots, w_k \in \Sigma^N$ we add a state $q_f^{w_1, \ldots, w_k}$. Let s_i be a stream term representing the word w_i, with a variable x_i at the tail, like in the proof of Theorem 5.2. Assume without loss of generality that the stream arguments of f occur before the data arguments, and let $y_1, \ldots, y_j$ be data variables. Let $l_1 \to r_1, \ldots, l_n \to r_n \in R$ be all rules such that $f(s_1, \ldots, s_k, y_1, \ldots, y_j)$ unifies with l_i with substitution σ_i. Let M be the maximum number of data arguments of any defined stream function symbol in R. We keep the representations of data arguments in constructor normal form on M separate work tapes: we call them argument work tapes.

Assume e.g. $k = 2$ and $w_1 = ab$ and $w_2 = cd$ and $j = 2$. In the state $q_f^{ab,cd}$ the JTT first checks which of the left-hand sides $l_1, \ldots, l_n$ matches $f(a :: b :: x_1, c :: d :: x_2, u_1, u_2)$ where u is the first data argument – the data term whose representation is stored on the first argument work tape. There is at most one matching l_i because R is orthogonal, and this can be checked using only logarithmic space (it suffices to check whether the two data arguments in l_i match u_1, u_2, respectively). If none of the l_i matches then the JTT loops. Assume e.g. l_i matches with substitution σ and

$$\sigma l_i = f(a :: b :: x_1, c :: d :: x_2, S(y), z)$$

and

$$\sigma r_i = h_1(a, b, y, z) :: g(c :: d :: x_2, b :: x_1, d :: x_2, h_2(y), y).$$

Then the JTT outputs the constructor normal form of $h_1(a, b, t_1, t_2)$, computed using Proposition 3.17, where $S(t_1)$ is the constructor normal form of the first data argument, stored on the first argument work tape, and t_2 is the constructor normal form of the second data argument, sored on the second argument work tape. If the constructor normal form of $h_1(a, b, t_1, t_2)$ is not in Σ, then the JTT loops. Next, the JTT simultaneously sets the first cursor to the second one, the second to the first, and the third to the second. Then it computes the constructor normal form of $h_2(t)$, using Proposition 3.17, and writes it to the first argument tape, and also copies t to the second argument tape. Next, the JTT reads one symbol from the second cursor and one from the third, moving them forward. Let these symbols be a_1, a_2 respectively. The JTT then enters the state q_g^{cd, ba_1, da_2}. This behaviour may always be encoded using a finite number of states.

The rest of the construction is like in the proof of Theorem 5.2.

It is clear that the constructed JTT computes a stream $|s'| \in \Sigma^{\leq \omega}$ for an $s' \in \mathcal{S}^+(\Sigma)$ such that $f_0(s_1, \ldots, s_n) \to_{R\perp e}^{\infty} s'$. As mentioned before, Lemma 6.4 and Lemma 6.5 imply that this is correct. Indeed, we have $f_0(s_1, \ldots, s_n) \to_R^{\infty} s$ where $F(|s_1|, \ldots, |s_n|) = |s|$. By Lemma 6.4 there is w with $s \to_{R\perp e}^{\infty} w$ and $s' \to_R^{\infty} w$. By Lemma 6.5 we have $w \in \mathcal{S}^+(\Sigma)$ and $s \sim w$ and $s' \sim w$. Thus $|s| = |w| = |s'|$. So the JTT computes the stream $|s|$, as required.   ◀

Decreasing Diagrams with Two Labels Are Complete for Confluence of Countable Systems

Jörg Endrullis

Vrije Universiteit Amsterdam,
Department of Computer Science,
Amsterdam, the Netherlands

Jan Willem Klop

Vrije Universiteit Amsterdam,
Department of Computer Science,
Amsterdam, the Netherlands
and
Centrum Wiskunde & Informatica (CWI),
Amsterdam, the Netherlands

Roy Overbeek

Vrije Universiteit Amsterdam,
Department of Computer Science,
Amsterdam, the Netherlands

Abstract

Like termination, confluence is a central property of rewrite systems. Unlike for termination, however, there exists no known complexity hierarchy for confluence. In this paper we investigate whether the decreasing diagrams technique can be used to obtain such a hierarchy. The decreasing diagrams technique is one of the strongest and most versatile methods for proving confluence of abstract reduction systems, it is complete for countable systems, and it has many well-known confluence criteria as corollaries.

So what makes decreasing diagrams so powerful? In contrast to other confluence techniques, decreasing diagrams employ a labelling of the steps $\to$ with labels from a well-founded order in order to conclude confluence of the underlying unlabelled relation. Hence it is natural to ask how the size of the label set influences the strength of the technique. In particular, what class of abstract reduction systems can be proven confluent using decreasing diagrams restricted to 1 label, 2 labels, 3 labels, and so on? Surprisingly, we find that two labels suffice for proving confluence for every abstract rewrite system having the cofinality property, thus in particular for every confluent, countable system. We also show that this result stands in sharp contrast to the situation for commutation of rewrite relations, where the hierarchy does not collapse.

Finally, as a background theme, we discuss the logical issue of first-order definability of the notion of confluence.

Figure 1 Confluence.

Figure 2 Commutation.

1 Introduction

A binary relation $\rightarrow$ is called *confluent* if two coinitial reductions (i.e., reductions having the same starting term) can always be extended to cofinal reductions, that is:

$$\forall abc. \left(b \twoheadleftarrow a \twoheadrightarrow c \Rightarrow \exists d. \ b \twoheadrightarrow d \twoheadleftarrow c \right) . \tag{1}$$

The confluence property is illustrated in Figure 1, in which solid and dotted lines stand for universal and existential quantification, respectively. The relation $\rightarrow$ is called *terminating* if there are no infinite sequences $a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots$.

Termination and confluence are central properties of rewrite systems. For both properties there exist numerous proof techniques, and there are annual competitions for comparing the performance of automated provers. It is therefore a natural question how to measure and classify the complexity of termination and confluence problems. While there is a well-known hierarchy for termination [20], no such classification is known for confluence.[1]

The termination hierarchy [20] is based on the characterisation of termination in terms of well-founded monotone algebras. This entails an interpretation of the symbols of the signature as functions over the algebra. Then the class of the functions (or other properties of the algebra) used to establish termination can serve as a measure for the complexity of the termination problem. For instance, if polynomial functions over the natural numbers suffice to establish termination, then the rewrite system is said to be polynomially terminating.

In order to address the question of a hierarchy and complexity measure for the confluence property, our point of departure is the decreasing diagrams technique [17]. Decreasing diagrams are for confluence what well-founded interpretations are for termination. The decreasing diagrams technique is complete for systems having the cofinality property [15, p. 766]. Thus, in particular for every confluent, countable abstract reduction system, the confluence property can be proven using the decreasing diagrams technique. The power of decreasing diagrams is moreover witnessed by the fact that many well-known confluence criteria are direct consequences of decreasing diagrams [17], including the lemma of Hindley–Rosen [6, 13], Rosen's request lemma [13], Newman's lemma [12], and Huet's strong confluence lemma [7].

What makes the decreasing diagrams technique so powerful? The freedom to label the steps distinguishes decreasing diagrams from all other confluence criteria, with the exception of the weak diamond property [1, 4] by De Bruijn which has equal strength. This suggests that the power of these techniques arises from the labelling. This naturally leads to the following questions:
1. How does the size of the label set influence the strength of decreasing diagrams?

---

[1] Ketema and Simonsen [8] consider peaks $t_1 \leftarrow s \twoheadrightarrow t_2$ and measure the length of joining reductions $t_1 \twoheadrightarrow \cdot \twoheadleftarrow t_2$ as a function of the size of s and the length of the reductions in the peak. The nature of this function can serve as a complexity measure for a confluence problem.

2. What class of abstract reduction systems can be proven confluent using decreasing diagrams with 1 label, 2 labels, 3 labels and so on?

3. Can the size of the label set serve as a complexity measure for a confluence problem?

Let DCR denote the class of abstract reduction systems (ARSs) whose confluence can be proven using decreasing diagrams. For an ordinal α, we write DCR_α for the class of ARSs whose confluence can be proven using decreasing diagrams with label set α (see Definition 15).

For every ARS $\mathcal{A}$, we have

$$DCR(\mathcal{A}) \implies DCR_\alpha(\mathcal{A}) \text{ for some ordinal } \alpha \tag{2}$$

The reason is that any partial well-founded order can be transformed into a total well-founded order (thus an ordinal). This transformation does not require the Axiom of Choice, see [4].

Clearly, we have $DCR_\alpha \subseteq DCR_\beta$ whenever $\alpha < \beta$. So

$$DCR_0 \subseteq DCR_1 \subseteq DCR_2 \subseteq DCR_3 \subseteq \ldots \subseteq DCR_\omega \subseteq \ldots \tag{3}$$

But which of these inclusions are strict? From the completeness proof in [18] it follows that all abstract reduction systems having the *cofinality property*, including all countable systems, belong to DCR_ω. In other words, for confluence of countable systems it suffices to label steps with natural numbers.

Contribution and outline

Our main result is that all systems with the cofinality property are in the class DCR_2, see Section 4. In particular, for proving confluence of countable abstract reduction systems it always suffices to label steps with 0 or 1 using the order $0 < 1$. So for countable systems, the hierarchy (3) collapses at level DCR_2. This is somewhat surprising, as one might expect that decreasing diagrams draws its strength from a rich labelling of the steps.

Interestingly, there is a stark contrast with commutation. For commutation the hierarchy does not collapse, see Section 5. We prove that, for commutation of countable systems, all inclusions are strict up to level DC_ω.

Our findings also provide new ways to approach the long-standing open problem of completeness of decreasing diagrams for uncountable systems, see Section 6.

2 Preliminaries

We repeat some of the main definitions, for the sake of self-containedness, and to fix notations. Let A be a set. For a relation $\to \subseteq A \times A$ we write $\to^*$ or $\twoheadrightarrow$ for its reflexive transitive closure. We write $\equiv$ for the empty step, that is, $\equiv = \{(a,a) \mid a \in A\}$, and we define $\to^\equiv = \to \cup \equiv$.

▶ **Definition 1** (Abstract Reduction System). An *abstract reduction system (ARS)* $\mathcal{A} = (A, \to)$ consists of a non-empty set A together with a binary relation $\to \subseteq A \times A$. For $B \subseteq A$ we define $\mathcal{A}|_B$, the *restriction of $\mathcal{A}$ to B*, by $\mathcal{A}|_B = (B, \to \cap (B \times B))$.

▶ **Definition 2** (Indexed ARS). An *indexed ARS* $\mathcal{A} = (A, \{\to_\alpha\}_{\alpha \in I})$ consists of a non-empty set A of *objects*, and a family $\{\to_\alpha\}_{\alpha \in I}$ of relations $\to_\alpha \subseteq A \times A$ indexed by some set I.

▶ **Definition 3** (Confluence). An ARS $(A, \to)$ is *confluent (CR)* if $\leftarrow \cdot \twoheadrightarrow \subseteq \twoheadrightarrow \cdot \twoheadleftarrow$, that is, every pair of finite, coinitial rewrite sequences can be joined to a common reduct.

▶ **Definition 4** (Commutation). Let $(A, \to, \rightsquigarrow)$ be an indexed ARS. Then the relation $\to$ *commutes with* $\rightsquigarrow$ if $\leftarrow^* \cdot \rightsquigarrow^* \subseteq \rightsquigarrow^* \cdot \leftarrow^*$; see Figure 2.

▶ **Definition 5** (Countable). An ARS $(A, \rightarrow)$ is *countable (CNT)* if there exists a surjective function from the set of natural numbers $\mathbb{N}$ to A.

▶ **Definition 6** (Cofinal Reduction). Let $\mathcal{A} = (A, \rightarrow)$ be an ARS. A set $B \subseteq A$ is *cofinal* in $\mathcal{A}$ if for every $a \in A$ we have $a \twoheadrightarrow b$ for some $b \in B$. A finite or infinite reduction sequence $b_0 \rightarrow b_1 \rightarrow b_2 \rightarrow \cdots$ is *cofinal* in $\mathcal{A}$ if the set $B = \{\, b_i \mid i \geq 0 \,\}$ is cofinal in $\mathcal{A}$.

▶ **Definition 7** (Cofinality Property). An ARS $\mathcal{A} = (A, \rightarrow)$ has the *cofinality property (CP)* if for every $a \in A$, there exists a reduction $a \equiv b_0 \rightarrow b_1 \rightarrow b_2 \rightarrow \cdots$ that is cofinal in $\mathcal{A}|_{\{b \,|\, a \twoheadrightarrow b\}}$.

▶ **Lemma 8.** *Let $\mathcal{A} = (A, \rightarrow)$ be a confluent ARS and $a \in A$. If a rewrite sequence is cofinal in $\mathcal{A}|_{\{b \,|\, a \twoheadrightarrow b\}}$, then it is also cofinal in $\mathcal{A}|_{\{b \,|\, a \leftrightarrow^* b\}}$.* ◀

▶ **Theorem 9** (Klop [9]). *Every confluent countable ARS has the cofinality property.* ◀

3 First-order Definability of Confluence

As we are investigating a confluence hierarchy, the question of first-order definability of confluence arises naturally. Namely, if confluence were definable by a set of first-order formulas, then we could obtain a confluence hierarchy by imposing syntactic restrictions on this set of formulas.

At first glance this question may appear trivial since confluence is typically defined via the first-order formula (1). However, this formula involves the transitive closure $\twoheadrightarrow$ of the one-step relation $\rightarrow$ which is itself not first-order definable. We show that confluence is not first-order definable over the one-step relation $\rightarrow$.

▶ **Remark.** In [16] it is shown that the first-order theory of linear one-step rewriting is undecidable. In this paper it is mentioned as a conjecture that undecidable properties like confluence and weak termination (see further [2]) cannot be expressed in the first-order logic of one-step rewriting.

▶ **Theorem 10.** *Confluence and local confluence cannot be defined in the first-order logic with equality and the predicate $\rightarrow$ (one-step rewriting), neither by a single formula nor by a set of formulas.*

Proof. Assume, for a contradiction, that there is a set Δ of first-order formulas over the predicate $\rightarrow$ such that for every ARS $\mathcal{A} = (A, \rightarrow)$ it holds that:

$$\mathcal{A} \text{ is confluent} \quad \Longleftrightarrow \quad \mathcal{A} \models \Delta$$

Here $\mathcal{A} \models \Delta$ means that $\mathcal{A}$ is a *model* of Δ, that is, $\mathcal{A}$ satisfies all formulas in Δ. In what follows, we write $[c]$ for the interpretation of a constant c. For convenience, we write $\rightarrow$ for the predicate symbol in formulas as well as for the actual one-step rewrite relation or $\mathcal{A}$.

Our goal is to describe the following non-confluent structure using formulas:
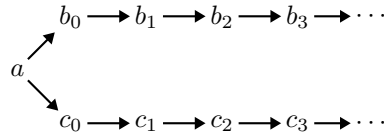
We start by describing each single step by a formula:

$$\Lambda = \{\, a \rightarrow b_0,\ a \rightarrow c_0 \,\} \cup \{\, b_i \rightarrow b_{i+1} \mid i \in \mathbb{N} \,\} \cup \{\, c_j \rightarrow c_{j+1} \mid j \in \mathbb{N} \,\}$$

We need to ensure that the interpretation of distinct constants is distinct:

$$\Lambda_{\neq} = \{\, x \neq y \mid x, y \in N \,\} \quad \text{where} \quad N = \{\, a \,\} \cup \{\, b_i \mid i \in \mathbb{N} \,\} \cup \{\, c_j \mid j \in \mathbb{N} \,\}$$

Finally, the following formula requires all elements, except for $[a]$, to be deterministic:

$$\xi = \forall xyz.\, (x \neq a \land x \to y \land x \to z) \Rightarrow y = z$$

This simple trick excludes that elements $\{[b_n] \mid n \in \mathbb{N}\} \cup \{[c_n] \mid n \in \mathbb{N}\}$ admit steps other than the ones specified in Λ.

Now consider the following set of formulas:

$$\Gamma = \Delta \cup \Lambda \cup \Lambda_{\neq} \cup \{\,\xi\,\}$$

By the above construction, any model of $\Lambda \cup \Lambda_{\neq} \cup \{\,\xi\,\}$ cannot be confluent. However, any model of Δ must be confluent. Thus Γ does not have a model.

On the other hand, any finite subset Γ' of Γ has a model. This can be seen as follows. There exists a $k \in \mathbb{N}$ such that none of the constants $\{\, b_i \mid i \geq k \,\} \cup \{\, c_j \mid j \geq k \,\}$ appears in Γ'. Then the following structure is a model of Γ':
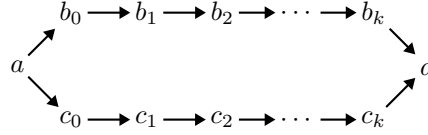
This is a contradiction! Due to the *compactness theorem*, Γ has a model if and only if every finite subset of Γ has a model. Thus confluence is not first-order definable.

Note that the same proof also shows undefinability of local confluence. ◄

▶ **Theorem 11.** *For $\alpha \geq 2$, DCR_α cannot be defined in the first-order logic with equality and the predicate $\to$ (one-step rewriting), neither by a single formula nor by a set of formulas.*

Proof. Follows by an extension of the proof for Theorem 10, noting that the model of Γ' admits a decreasing labelling with 2 labels. ◄

Note that DCR_1 is equivalent to the diamond property for the reflexive closure of the rewrite relation, and thus is first-order definable.

4 Decreasing Diagrams for Confluence with Two Labels

In this section we show that two labels suffice for proving confluence using decreasing diagrams for any abstract reduction system having the cofinality property. We start by introducing the decreasing diagrams technique.

▶ **Notation 12.** For an indexed ARS $\mathcal{A} = (A, \{\to_\alpha\}_{\alpha \in I})$ and a relation $< \subseteq I \times I$, we define

$$\to \;=\; \bigcup_{\alpha \in I} \to_\alpha \qquad\qquad \to_{<\beta} \;=\; \bigcup_{\alpha < \beta} \to_\alpha \qquad\qquad \to_{\leq \beta} \;=\; \bigcup_{\alpha \leq \beta} \to_\alpha$$

Moreover, we use $\to_{<\alpha \cup <\beta}$ as shorthand for $(\to_{<\alpha} \cup \to_{<\beta})$.

▶ **Definition 13** (Decreasing Church–Rosser [17]). An ARS $\mathcal{A} = (A, \rightsquigarrow)$ is called *decreasing Church–Rosser (DCR)* if there exists an ARS $\mathcal{B} = (A, \{\to_\alpha\}_{\alpha \in I})$ indexed by a well-founded partial order $(I, <)$ such that $\rightsquigarrow \;=\; \to$ and every peak $c \leftarrow_\beta a \to_\alpha b$ can be joined by reductions of the form shown in Figure 3.[2]
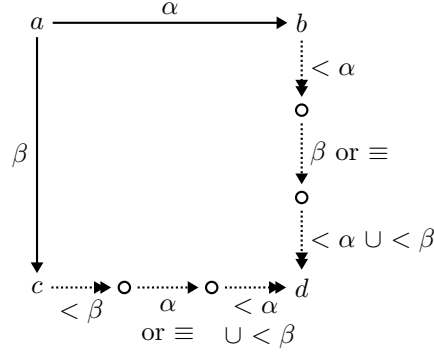
🟨 **Figure 3** Decreasing elementary diagram.

The following is the main theorem of decreasing diagrams.

▶ **Theorem 14** (Decreasing Diagrams – De Bruijn [1] & Van Oostrom [17]). *If an ARS is decreasing Church–Rosser, then it is confluent.*   ◀

In other words $DCR \implies CR$.

As already suggested in the introduction, we introduce classes DCR_α restricting the well-founded order $(I, <)$ in Definition 13 to the ordinal α.

▶ **Definition 15.** For ordinals α, let DCR_α denote the class of ARSs $\mathcal{A}$ that are decreasing Church–Rosser (Definition 13) with label set $\{\beta \mid \beta < \alpha\}$ ordered by the usual order $<$ on ordinals. We say that $\mathcal{A}$ has the property DCR_α, denoted $DCR_\alpha(\mathcal{A})$, if $\mathcal{A} \in DCR_\alpha$.

The remainder of this section is devoted to the proof that every system with the cofinality property is DCR_2. Put differently, it suffices to label steps with $I = \{0, 1\}$. Let $\mathcal{A} = (A, \rightarrow)$ be an ARS having the cofinality property. Note that, for defining the labelling, we can consider connected components with respect to $\leftrightarrow^*$ separately. Thus assume that $\mathcal{A}$ consists of a single connected component, that is, for every $a, b \in A$ we have $a \leftrightarrow^* b$. By the cofinality property, which implies confluence, and Lemma 8 there exists a rewrite sequence

$$m_0 \rightarrow m_1 \rightarrow m_2 \rightarrow m_3 \rightarrow \cdots$$

that is cofinal in $\mathcal{A}$; we call this rewrite sequence the *main road*. Without loss of generality we may assume that the main road is acyclic, that is, $m_i \not\equiv m_j$ whenever $i \neq j$. (We can eliminate loops without harming the cofinality property. Note that the main road is allowed to be finite.)

The idea of labelling the steps in $\mathcal{A}$ is as follows. For every node $a \in A$, we label precisely one of the outgoing edges with 0 and all others with 1. The edge labelled with 0 must be part of a shortest path from a to the main road. For the case that a lies on the main road, the step labelled 0 must be the step on the main road. This is illustrated in Figure 4.

Note that there is a choice about which edge to label with 0 whenever there are multiple outgoing edges that all start a shortest path to the main road. To resolve this choice, the

---

[2] Van Oostrom [19] generalises the shape of the decreasing elementary diagrams by allowing the joining reductions to be conversions. This can be helpful to find suitable elementary diagrams. However, if there are conversions then we can always obtain joining reductions by diagram tiling. So a system is locally decreasing with respect to conversions if and only if it is locally decreasing with respect to reductions (using the same labelling of the steps).
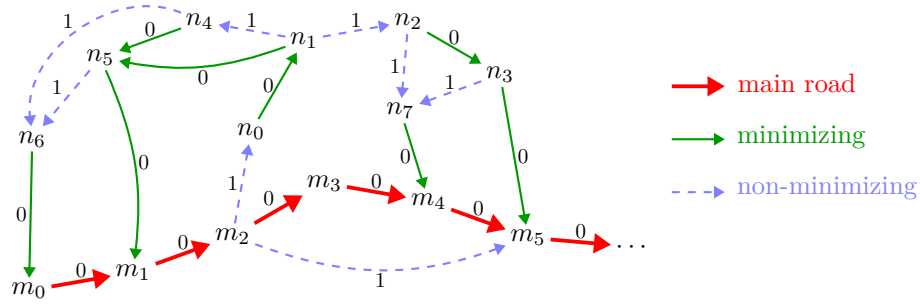
Figure 4 Example labelling.

following definition assumes a well-order $<$ on the universe A, whose existence is guaranteed by the well-ordering theorem. Then, whenever there is a choice, we choose the edge for which the target is minimal in this order.

▶ Remark. Recall that the Axiom of Choice is equivalent to the well-ordering theorem. In many practical cases, however, the existence of such a well-order does not require the Axiom of Choice. If the universe is countable, then such a well-order can be derived directly from the surjective counting function $f : \mathbb{N} \to A$.

In the following definition we follow the proof in [15, Proposition 14.2.30, p. 766], employing the notion of a cofinal sequence and the rewrite distance from a point to this sequence. While the proof in [15] labels steps by their distance to the target node, we need a more sophisticated labelling.

▶ **Definition 16.** Let $\mathcal{A} = (A, \to)$ be an ARS and $M : m_0 \to m_1 \to m_2 \to \cdots$ be a finite or infinite rewrite sequence in $\mathcal{A}$. For $a, b \in A$, we write

(i) $a \in M$ if $a \equiv m_i$ for some $i \geq 0$, and

(ii) $(a \to b) \in M$ if $a \equiv m_i$ and $b \equiv m_{i+1}$ for some $i \geq 0$.

If M is cofinal in $\mathcal{A}$, we define the *distance* $d(a, M)$ as the least natural number $n \in \mathbb{N}$ such that $a \to^n m$ for some $m \in M$. If M is clear from the context, we write $d(a)$ for $d(a, M)$.

▶ **Definition 17** (Labelling with two labels). Let $\mathcal{A} = (A, \to)$ be an ARS equipped with a well-order $<$ on A such that there exists a cofinal reduction $M : m_0 \to m_1 \to m_2 \to \cdots$ that is acyclic (that is, for all $i < j$, $m_i \not\equiv m_j$).

We say that a step $a \to b$ is

(i) *on the main road* if $(a \to b) \in M$;

(ii) *minimizing* if $d(a) = d(b) + 1$ and $b' \geq b$ for every $a \to b'$ with $d(b') = d(b)$.

We define an indexed ARS $\mathcal{A}_{\{0,1\}} = (A, \{\to_i\}_{i \in I})$ where $I = \{0, 1\}$ as follows:

$$a \to_0 b \iff a \to b \text{ and this step is on the main road or minimizing}$$

$$a \to_1 b \iff a \to b \text{ and this step is not on the main road and not minimizing}$$

for every $a, b \in A$.

▶ **Lemma 18.** *Let $\mathcal{A} = (A, \to)$ be an ARS with a cofinal rewrite sequence $M : m_0 \to m_1 \to \cdots$ that is acyclic. Furthermore, let $<$ be a well-order over A. Then for $\mathcal{A}_{\{0,1\}} = (A, \to_0, \to_1)$ we have:*

(i) $\to \; = \; \to_0 \cup \to_1$;

(ii) *for every $a, b \in M$ we have $a \twoheadrightarrow_0 \cdot {}_0\!\twoheadleftarrow b$;*

(iii) *for every $a \in A$, there is at most one $b \in A$ such that $a \to_0 b$;*

(iv) *for every $a \notin M$, there exists $b \in A$ with $a \to_0 b$ and $d(a) > d(b)$;*
(v) *for every $a \in A$, there exists $m \in M$ such that $a \twoheadrightarrow_0 m$;*
(vi) *every peak $c \leftarrow_\beta a \to_\alpha b$ can be joined as in Figure 3, and, explicitly for labels $\{0, 1\}$, as in Figure 5.*

Proof. Properties i and ii follow from the definitions.

For iii assume that $b \leftarrow_0 a \to_0 c$. We show that $b \equiv c$. The steps $a \to b$ and $a \to c$ are either minimizing or on the main road. We distinguish cases $a \in M$ and $a \notin M$:

(i) Assume that $a \in M$. Then $d(a) = 0$, and thus neither $a \to b$ nor $a \to c$ is a minimizing step. Hence $(a \to b) \in M$ and $(a \to c) \in M$. Since M is acyclic, we get $b \equiv c$.

(ii) If $a \notin M$, both steps $a \to b$ and $a \to c$ must be minimizing. If $d(b) \neq d(c)$, then we have either $d(a) \neq d(b) + 1$ or $d(a) \neq d(c) + 1$, contradicting minimization. Thus $d(b) = d(c)$. Then by minimization we have $b \geq c$ and $c \geq b$, from which we obtain $b \equiv c$.

For iv, consider an element $a \notin M$. Let $B = \{b' \mid a \to b' \wedge d(a) = d(b') + 1\}$. By definition of the distance $d(\cdot)$, $B \neq \varnothing$. Define b as the least element of B in the well-order $<$ on A. It follows that $a \to b$ is a minimization step. Hence $a \to_0 b$ and $d(a) > d(b)$. Property v follows directly from iv using induction on the distance.

For vi, consider a peak $c \leftarrow_\beta a \to_\alpha b$. If $b \equiv c$, then the joining reductions are empty steps. Thus assume that $b \not\equiv c$. By iii we have either $\alpha = 1$ or $\beta = 1$. By v there exist $m_b, m_c \in M$ such that $b \twoheadrightarrow_0 m_b$ and $c \twoheadrightarrow_0 m_c$. By ii we have $m_b \twoheadrightarrow_0 \cdot \twoheadleftarrow_0 m_c$. Hence $b \twoheadrightarrow_0 \cdot \twoheadleftarrow_0 c$. These joining reductions are of the form required by Figure 3 since $\twoheadrightarrow_0 = \twoheadrightarrow_{<\alpha \cup <\beta}$.   ◀

▶ **Theorem 19.** *If an ARS $\mathcal{A} = (A, \to)$ satisfies the cofinality property, then there exists an indexed ARS $(A, (\to_\alpha)_{\alpha \in \{0,1\}})$ such that $\to = \to_0 \cup \to_1$ and every peak $c \leftarrow_\beta a \to_\alpha b$ can be joined according to the elementary decreasing diagram in Figure 3, and, explicitly for labels $\{0, 1\}$, as in Figure 5.*

Proof. It suffices to consider a connected component of $\mathcal{A}$. Let $\mathcal{B} = (B, \to)$ be a connected component of $\mathcal{A}$: we have $a \leftrightarrow^* b$ for all $a, b \in B$. By the cofinality property and Lemma 8, there exists a cofinal reduction $m_0 \to m_1 \to \cdots$ in $\mathcal{B}$. By the well-ordering theorem, there exists a well-order $<$ over B. Then $\mathcal{B}$ has the required properties by Lemma 18vi.   ◀

▶ **Corollary 20.** *DCR_2 is a complete method for proving confluence of countable ARSs.*

Proof. Immediate from Theorems 9 and 19.   ◀

Theorem 19 also holds for De Bruijn's weak diamond property. Note the following caveat: when restricting the index set I to a single label, the decreasing diagram technique is equivalent to $\leftarrow \cdot \to \, \subseteq \, \to^\equiv \cdot \leftarrow^\equiv$, i.e. the *diamond property* for $\to \cup \equiv$, while the weak diamond property with one label is equivalent to *strong confluence* $\leftarrow \cdot \to \, \subseteq \, \to^\equiv \cdot \leftarrow$.

The property DCR_2 is given implicitly by the decreasing diagrams as in Figure 3, but it is also instructive to give explicitly the elementary reduction diagrams making up the property DCR_2. These are shown in Figure 5. Note that the 1-steps do not split in the diagram construction, i.e. they cross over in at most one copy. This facilitates a simple proof of confluence.

Actually, from our proof it follows that the joining reductions can be required to only contain steps with label 0. Thus even the simple shape of diagrams shown in Figure 6 is complete for proving confluence of systems having the cofinality property. Here the 1-steps do not cross over at all! Note that while this set of elementary diagrams has a trivial proof of confluence, the work to prove $DCR_2 \implies CR$ from the original elementary diagrams as in Figure 5, consists in showing from our earlier construction that it actually suffices to join by using only 0's.
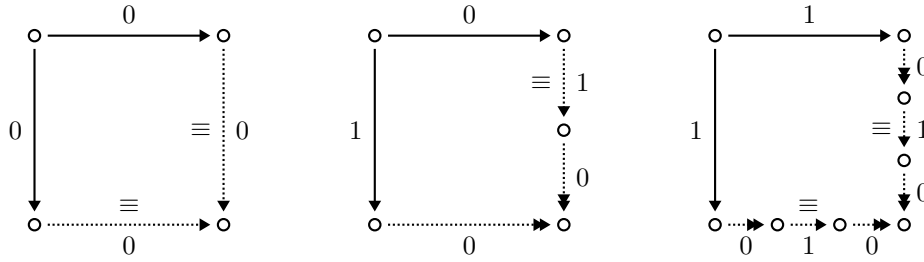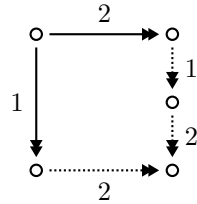
Figure 5 Decreasing diagrams with labels 0 and 1 where $0 < 1$.
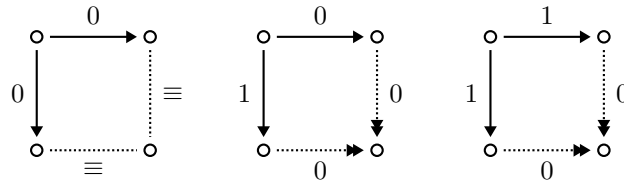
Figure 6 A simple set of diagrams that is complete for confluence of countable systems.

▶ Remark. We note a certain similarity between the notion of a decreasing diagram based on labels $\{0, 1\}$ with $0 < 1$ and the classical 'requests' lemma of J. Staples [10, 15, Exercise 2.08.5, p. 9]. In $\mathcal{A} = (A, \to_1, \to_2)$ define: $\to_1$ requests $\to_2$ if

If in addition $\to_1$ and $\to_2$ are confluent, then $\to_{1,2} = \to_1 \cup \to_2$ is confluent.

The requests lemma states that the 'dominant' reduction $\twoheadrightarrow_1$ needs the 'support' of the secondary reduction $\twoheadrightarrow_2$ for making the divergence $\twoheadleftarrow_1 \cdot \twoheadrightarrow_2$ convergent. Similarly for the property DCR_2, the dominant reduction $\to_1$ needs support by $\twoheadrightarrow_0$ for making the divergence $\leftarrow_1 \cdot \to_0$ convergent. However, the requests lemma employs $\twoheadrightarrow$, not $\to$.

5 Decreasing Diagrams for Commutation

The decreasing diagram technique can also be used for proving commutation, see [17]. It turns out that the situation for commutation stands in sharp contrast to that for confluence. For commutation the hierarchy does not collapse. In particular, we show that, for every $n \leq \omega$, decreasing diagrams for commutation with n labels is *strictly* stronger than decreasing diagrams with less than n labels.

The elementary decreasing diagram for commutation is shown in Figure 7, which is very similar to Figure 3, but now refers to two 'basis' relations $\to$, $\rightsquigarrow$.

▶ **Definition 21** (Decreasing Commutation). An ARS $\mathcal{A} = (A, \to, \rightsquigarrow)$ is called *decreasing commuting (DC)* if there is an ARS $\mathcal{B} = (A, \{\to_\alpha\}_{\alpha \in I}, \{\rightsquigarrow_\alpha\}_{\alpha \in I})$ indexed by a well-founded partial order $(I, <)$ such that $\to_\mathcal{A} = \to_\mathcal{B}$ and $\rightsquigarrow_\mathcal{A} = \rightsquigarrow_\mathcal{B}$, and every peak $c \leftarrow_\beta a \rightsquigarrow_\alpha b$ in $\mathcal{B}$ can be joined by reductions of the form shown in Figure 7.
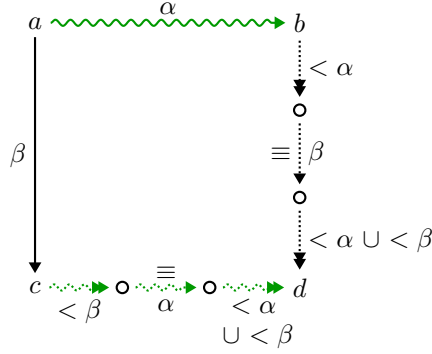
Figure 7 Decreasing elementary diagram for proving commutation.

If all conditions are fulfilled, we call $\mathcal{B}$ a *decreasing labelling* of $\mathcal{A}$.

▶ **Theorem 22** (Decreasing Diagrams for Commutation – Van Oostrom [17]). *If an ARS $\mathcal{A} = (A, \rightarrow, \rightsquigarrow)$ is decreasing commuting, then $\rightarrow$ commutes with $\rightsquigarrow$.*   ◀

Analogous to the classes DCR_α for confluence, we introduce classes DC_α for commutation.

▶ **Definition 23.** For ordinals α, let DC_α denote the class of ARSs $\mathcal{A} = (A, \rightarrow, \rightsquigarrow)$ that are decreasing commuting (Definition 21) with label set $\{\,\beta \mid \beta < \alpha\,\}$ ordered by the usual order $<$ on ordinals. We say that $\mathcal{A}$ has the property DC_α, denoted $DC_\alpha(\mathcal{A})$, if $\mathcal{A} \in DC_\alpha$.

In Definition 23 it suffices to consider total orders since every partial well-founded order can be transformed into a total well-founded order. This transformation [4] preserves the decreasing elementary diagrams and does not need the Axiom of Choice.

In order to show that the hierarchy for commutation does not collapse, we inductively construct, for every $n \in \mathbb{N}$, an ARS $\mathcal{A}_n$ that is DC_{5n+1}, but not DC_n.

▶ **Definition 24.** For every $n \in \mathbb{N}$ we define a tuple $\Phi_n = (\mathcal{A}_n, a_1, a, c, b, b_1)$ consisting of an ARS $\mathcal{A}_n = (A_n, \rightarrow_n, \rightsquigarrow_n)$ and distinguished elements $a_1, a, c, b, b_1 \in A_n$ by induction on n:
1. Let $\Phi_0 = (\mathcal{A}_0, a_1, c, c, c, b_1)$ where $\mathcal{A}_0$ is the ARS displayed in Figure 8.
2. Let $\Phi_n = (\mathcal{A}_n, a, a', c, b', b)$. We obtain $\mathcal{A}_{n+1}$ as an extension of $\mathcal{A}_n$ as shown in Figure 9. The inner dark part with the darker background is $\mathcal{A}_n$. The extension consists of the addition of fresh elements $a_1, \ldots, a_7$ and $b_1, \ldots, b_7$ and rewrite steps as shown in the figure. We define $\Phi_{n+1} = (\mathcal{A}_{n+1}, a_1, a, c, b, b_1)$.

We start with a few important properties of the construction.

▶ **Lemma 25.** *For every $n \in \mathbb{N}$ and $\Phi_n = (\mathcal{A}_n, a_1, a, c, b, b_1)$ with $\mathcal{A}_n = (A_n, \rightarrow, \rightsquigarrow)$ we have the following properties:*
 (i) *The relations $\rightarrow$ and $\rightsquigarrow$ are deterministic.*
 (ii) *For every element $x \in A_n$ we have $x \rightarrow^* c$ and $x \rightsquigarrow^* c$.*
 (iii) *For $x \in A_n$, we have $a_1 \rightsquigarrow^* x \leftarrow^* b_1$ if and only if $a \rightsquigarrow^* x$ and $a \rightarrow^* x$.*
 (iv) *For $x \in A_n$, we have $a_1 \rightarrow^* x \leftsquigarrow^* b_1$ if and only if $b \rightsquigarrow^* x$ and $b \rightarrow^* x$.*

Proof. We use induction on $n \in \mathbb{N}$. For the base case $n = 0$, we have $\Phi_0 = (\mathcal{A}_0, a_1, c, c, c, b_1)$ where $\mathcal{A}_0$ is given in Figure 8. The properties follow from an inspection of the figure.

For the induction step, let $n \in \mathbb{N}$ and assume that $\Phi_n = (\mathcal{A}_n, a, a', c, b', b)$ satisfies the properties. By construction, $\mathcal{A}_{n+1}$ is an extension of $\mathcal{A}_n$ as shown in Figure 9, and we have $\Phi_{n+1} = (\mathcal{A}_{n+1}, a_1, a, c, b, b_1)$. The fresh elements introduced by the extension are $X = \{\, a_1, \ldots, a_7, b_1, \ldots, b_7 \,\}$. We check the validity of each property for $\mathcal{A}_{n+1}$:
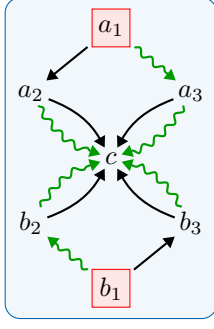
Figure 8 Base case: one label suffices.
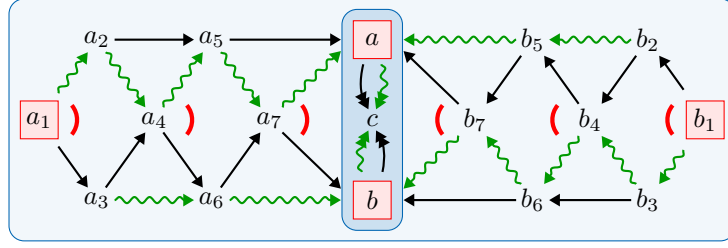
Figure 9 From n to $n+1$ labels for commutation. Rough proof sketch: Assume that at least one of the reductions $a \to^* c$, $b \rightsquigarrow^* c$, $a \rightsquigarrow^* c$ or $b \to^* c$ contains two steps labelled with n. Then each of the peaks at a_1, a_4 and a_7, or each of the peaks at b_1, b_4 and b_7 must contain a step labelled with $n+1$. As a consequence, one of the reductions $a_1 \to^* c$, $b_1 \rightsquigarrow^* c$, $a_1 \rightsquigarrow^* c$ or $b_1 \to^* c$ contains two steps labelled with $n+1$.

(i) There are no fresh steps with sources in $\mathcal{A}_n$. Every element $x \in X$ admits precisely one outgoing step $\to$ and one outgoing step $\rightsquigarrow$. So both rewrite relations remain deterministic, establishing property i.

(ii) For every element $x \in X$ we have $x \to^* a$ or $x \to^* b$, and $x \rightsquigarrow^* a$ or $x \rightsquigarrow^* b$. Together with the induction hypothesis ii for n, this yields property ii for $n+1$.

(iii) From Figure 9 it follows immediately that any reduction $a_1 \rightsquigarrow^* x \leftarrow^* b_1$ must be of the form $a_1 \rightsquigarrow^* a \rightsquigarrow^* x \leftarrow^* a \leftarrow^* b_1$. The reductions from both sides are deterministic and the first joining element is a.

(iv) Analogous to property iii.   ◀

From Lemma 25 ii it follows that $\to$ and $\rightsquigarrow$ commute in $\mathcal{A}_n$. However, commutation is not sufficient to conclude that $\mathcal{A}_n$ is decreasing commuting. Decreasing diagrams are not complete for proving commutation as shown in [4].

   We prove that $\mathcal{A}_n$ is decreasing commuting by constructing a labelling with $5n$ labels. This bound is by no means optimal, but easy to verify and sufficient for our purpose.

▶ **Lemma 26.** *For every $n \in \mathbb{N}$, $\mathcal{A}_n$ is DC_{5n+1}.*

Proof. We use induction on $n \in \mathbb{N}$. For the base case $n = 0$, consider $\mathcal{A}_0$ shown in Figure 8. For this system a single label suffices since the joining reductions in the elementary diagrams have length at most 1.

   For the induction step, assume that $\mathcal{A}_n$ has the property DC_{5n+1}. So $\mathcal{A}_n$ is decreasing commuting with labels $\{0, \ldots, \ell\}$ where $\ell = 5n$. By construction, $\mathcal{A}_{n+1}$ is an extension of $\mathcal{A}_n$ as shown in Figure 9. We extend the labelling of $\mathcal{A}_n$ with labels $\{0, \ldots, \ell\}$ to a labelling of $\mathcal{A}_{n+1}$ with labels $\{0, \ldots, \ell+5\}$ as follows:
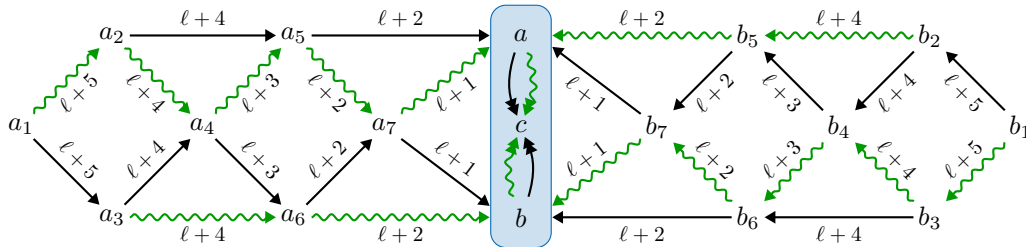
Here $\mathcal{A}_n$ is the darker inner part. From the picture it is easy to verify that every peak $\leftarrow \cdot \rightsquigarrow$ in the extension can be joined by reductions that only contain labels strictly smaller than labels of the peak. As a consequence, $\mathcal{A}_{n+1}$ is $DC_{5(n+1)+1}$.   ◀

Next, we show that $\mathcal{A}_n$ does not admit a decreasing labelling with n labels.

▶ **Lemma 27.** *For every $n \in \mathbb{N}$, $\mathcal{A}_n$ is not DC_n.*

Proof. We prove the following stronger claim: for every $n \in \mathbb{N}$ and $\Phi_n = (\mathcal{A}_n, a_1, a, c, b, b_1)$, and every decreasing labelling of $\mathcal{A}_n$ with labels from $\mathbb{N}$ it holds that at least one of the four paths $a_1 \to^* b$, $a_1 \rightsquigarrow^* a$, $b_1 \to^* a$ or $b_1 \rightsquigarrow^* b$ contains two labels $\geq n$. Note that these paths exist by Lemma 25. We prove this claim by induction on $n \in \mathbb{N}$.

For the base case $n = 0$, we have $\Phi_0 = (\mathcal{A}_0, a_1, c, c, c, b_1)$ where $\mathcal{A}_0$ is given in Figure 8. It suffices to consider one of the four paths. For instance, the rewrite sequence $a_1 \to^* c$ has length 2 and both steps must have a label ≥ 0.

For the induction step, assume that the claim holds for n and $\Phi_n = (\mathcal{A}_n, a, a', c, b', b)$. Accordingly, the induction hypothesis is that, for every decreasing labelling of $\mathcal{A}_n$ with labels from $\mathbb{N}$, one of the four paths $a \to^* b'$, $a \rightsquigarrow^* a'$, $b \to^* a'$ or $b \rightsquigarrow^* b'$ contains two labels $\geq n$. We prove the claim for $n + 1$. Let $\Phi_{n+1} = (\mathcal{A}_{n+1}, a_1, a, c, b, b_1)$ where $\mathcal{A}_{n+1}$ is an extension of $\mathcal{A}_n$ as shown in Figure 9. Let $\mathcal{B}$ be a decreasing labelling of the steps in $\mathcal{A}_{n+1}$ with labels from $\mathbb{N}$. We show that at least one of the paths $a_1 \to^* b$, $a_1 \rightsquigarrow^* a$, $b_1 \to^* a$ or $b_1 \rightsquigarrow^* b$ contains two labels $\geq n + 1$.

By construction, the systems $\mathcal{A}_{n+1}$ and $\mathcal{A}_n$ contain the same steps with sources in $\mathcal{A}_n$. Thus the restriction of the labelling $\mathcal{B}$ to $\mathcal{A}_n$ is a decreasing labelling for $\mathcal{A}_n$. By the induction hypothesis, at least one of the paths (i) $a \to^* b'$, (ii) $a \rightsquigarrow^* a'$, (iii) $b \to^* a'$ or (iv) $b \rightsquigarrow^* b'$ contains two labels $\geq n$. Without loss of generality, by symmetry, assume that the path (i) or (iv) contain two labels $\geq n$.

Consider the peak $a_3 \leftarrow a_1 \rightsquigarrow a_2$. As visible in Figure 9, every elementary diagram for this peak must have joining reductions of the form $a_3 \rightsquigarrow^* b \rightsquigarrow^* x \leftarrow^* a \leftarrow^* a_2$ for some $x \in \mathcal{A}_n$. From Lemma 25 iv we conclude that the joining reductions must be of the form

$$a_3 \rightsquigarrow^* b \rightsquigarrow^* b' \rightsquigarrow^* x \leftarrow^* b' \leftarrow^* a \leftarrow^* a_2$$

The path (i) $a \to^* b'$ or (iv) $b \rightsquigarrow^* b'$ contains two labels $\geq n$. Thus, for the elementary diagram to be decreasing, one of the steps in the peak $a_3 \leftarrow a_1 \rightsquigarrow a_2$ must have label $\geq n+1$.

The same argument can be applied to the peaks $a_6 \leftarrow a_4 \rightsquigarrow a_5$ and $b \leftarrow a_7 \rightsquigarrow a$. As a consequence, each of the peaks $a_3 \leftarrow a_1 \rightsquigarrow a_2$, $a_6 \leftarrow a_4 \rightsquigarrow a_5$ and $b \leftarrow a_7 \rightsquigarrow a$ contains one step with a label $\geq n + 1$. Hence at least one of the paths
1. $a_1 \to a_3 \to a_4 \to a_6 \to a_7 \to b$, or
2. $a_1 \rightsquigarrow a_2 \rightsquigarrow a_4 \rightsquigarrow a_5 \rightsquigarrow a_7 \rightsquigarrow a$

contains two steps with labels $\geq n + 1$.

If path (ii) $a \rightsquigarrow^* a'$ or (iii) $b \to^* a'$ contains two labels $\geq n$, then an analogous argument can be applied to the peaks $b_2 \leftarrow b_1 \rightsquigarrow b_3$, $b_5 \leftarrow b_4 \rightsquigarrow b_6$ and $a \leftarrow b_7 \rightsquigarrow b$, yielding that at least one of the paths $b_1 \to^* a$ or $b_1 \rightsquigarrow^* b$ contains two steps with labels $\geq n + 1$.

This proves the claim and concludes the proof. ◀

We have seen that, for every $n \in \mathbb{N}$, $\mathcal{A}_n$ that is DC_{5n+1}, but not DC_n (Lemmas 26 & 27). From this we can conclude that an infinite number of the inclusions $DC_0 \subseteq DC_1 \subseteq DC_2 \subseteq \cdots$ are strict. The following proposition allows us to infer that all of them are strict.

Roughly speaking, the following proposition states that if a level $\alpha + 1$ of the hierarchy does not collapse, then also the level α does not collapse. We state the proposition for the commutation hierarchy, but it also holds for the confluence hierarchy.

▶ **Proposition 28.** *If $DC_\alpha \subsetneq DC_{\alpha+1}$ for an ordinal α, then $DC_\beta \subsetneq DC_\alpha$ for every $\beta < \alpha$. This also holds when the classes are restricted to countable systems.*

Proof. Let $\mathcal{A} = (A, \to, \rightsquigarrow)$ be in $DC_{\alpha+1} \setminus DC_\alpha$. Then there exists a decreasing labelling $\mathcal{B}$ of $\mathcal{A}$ with labels $\{\,\beta \mid \beta \le \alpha\,\}$. As $\mathcal{A}$ is not DC_α some steps must have the maximum label α. Note that

$\star$ If the joining reductions in a decreasing elementary diagram contain a step with label α, then the corresponding peak must also contain a step with label α.

Let $\mathcal{B}'$ be obtained from $\mathcal{B}$ by dropping all steps with label α, and let $\mathcal{A}'$ be obtained from $\mathcal{B}'$ by dropping the labels. By $(\star)$, $\mathcal{B}'$ is a decreasing labelling of $\mathcal{A}'$, and hence $\mathcal{A}'$ is DC_α.

For a contradiction, assume that $DC_\beta = DC_\alpha$ for some $\beta < \alpha$. Then $\mathcal{A}'$ is DC_β. Let $\mathcal{B}''$ be obtained from $\mathcal{B}'$ by adding all steps that we had previously removed from $\mathcal{B}$, but we now relabel the steps from α to β. It is straightforward to check that $\mathcal{B}''$ is a decreasing labelling of $\mathcal{A}$. Hence, $\mathcal{A}$ is in $DC_{\beta+1} \subseteq DC_\alpha$. This is a contradiction. ◀

▶ **Example 29.** Assume that α is a limit ordinal and $DC_{\alpha+3} \subsetneq DC_{\alpha+4}$. By Proposition 28 we conclude $DC_{\alpha+2} \subsetneq DC_{\alpha+3}$. By repeated application of Proposition 28 we conclude

$$DC_\beta \subsetneq DC_\alpha \subsetneq DC_{\alpha+1} \subsetneq DC_{\alpha+2} \subsetneq DC_{\alpha+3} \subsetneq DC_{\alpha+4}$$

for every $\beta < \alpha$. However, the proposition does not help to conclude that $DC_\beta \subsetneq DC_{\beta'}$ for every $\beta < \beta' \le \alpha$.

▶ **Theorem 30.** *We have*

(i) $DC_n \subsetneq DC_{n+1}$ *for every $n \in \mathbb{N}$, and*

(ii) $\bigcup_{n \in \mathbb{N}} DC_n \subsetneq DC_\omega$.

These inclusions are strict also when the classes are restricted to countable systems.

Proof. By Lemmas 26 and 27 we know that $DC_n \subsetneq DC_{n+1}$ for infinitely many $n \in \mathbb{N}$. Then repeated application of Proposition 28 yields $DC_n \subsetneq DC_{n+1}$ for every $n \in \mathbb{N}$.

Let $\mathcal{A}$ be the infinite disjoint union $\mathcal{A}_0 \uplus \mathcal{A}_1 \uplus \mathcal{A}_2 \uplus \cdots$. As a consequence of Lemmas 26 and 27 the ARS $\mathcal{A}$ is DC_ω but not DC_n for any $n \in \mathbb{N}$. ◀

6 Conclusion

We study how the strength of decreasing diagrams is influenced by the size of the label set. We find that all abstract reduction systems with the cofinality property (in particular, all confluent, countable systems) can be proven confluent using the decreasing diagrams technique with the almost trivial label set $I = \{\,0, 1\,\}$. So for confluence of *countable* ARSs, we have the following implications:

$$CP \longrightarrow DCR_2 \longrightarrow DCR \longrightarrow CR$$

This is in sharp contrast to the situation for commutation for which we prove

$$DC_0 \subsetneq DC_1 \subsetneq DC_2 \subsetneq DC_3 \subsetneq \cdots \subsetneq DC_\omega$$

even for countable systems. So for commutation, for every $n \le \omega$, there exists a system that requires n labels. The structure of this hierarchy above level DC_ω remains open.

▶ **Open Problem 31.** *What inclusions $DC_\alpha \subseteq DC_\beta$ are strict for $\omega \le \alpha < \beta$?*

Decreasing diagrams are complete for confluence of countable systems. However, it is a long-standing open problem whether the method of decreasing diagrams is also complete for proving confluence of uncountable systems [17]. Our observations provide new ways for approaching this problem. In particular, it may be helpful to investigate the following:

▶ **Open Problem 32.** *Is there a confluent, uncountable system that is CR but not DCR_2?*

▶ **Open Problem 33.** *Is there a confluent, uncountable system that needs more than 2 labels to establish confluence using decreasing diagrams? In other words, is there an uncountable system that is DCR but not DCR_2? Is there an uncountable system that is DCR_3 but not DCR_2?*

So we have the following situation for uncountable systems:

$$CP \xrightarrow{\text{fails}^3} DCR_2 \xrightarrow{\text{new open}} DCR \xrightarrow{\text{open}} CR$$
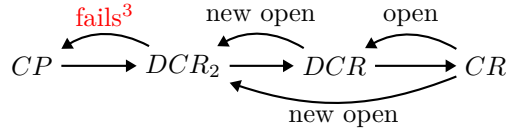
For a better understanding of this hierarchy, it would be interesting to investigate whether Proposition 28 can be generalised as follows.

▶ **Open Problem 34.** *Assume that $DC_\alpha \subsetneq DC_\beta$ for ordinals $\alpha < \beta$. Does this imply that none of the lower levels of the hierarchy collapse? That is, does it imply that $DC_{\alpha'} \subsetneq DC_{\beta'}$ for every $\alpha' < \beta' \leq \alpha$?*

Our findings indicate that the size of the label set in decreasing diagrams is not a suitable measure for the complexity of a confluence problem. So the complexity arises rather from the distribution of the labels, and the proof that every peak has suitable joining reductions. The complexity of the label distribution can be measured in terms of the complexity of machine required for computing the labels. For this purpose, one can consider Turing machines, finite automata or finite state transducers. The complexity of Turing machines can be measured in terms of time or space complexity, Kolmogorov Complexity [11] or degrees of unsolvability [14]. For finite state transducers the complexity can be classified by degrees of transducibility [5, 3].

──── **References** ────

1   N.G. de Bruijn. A Note on Weak Diamond Properties. Memorandum 78–08, Eindhoven Uninversity of Technology, 1978.

2   J. Endrullis, H. Geuvers, J.G. Simonsen, and H. Zantema. Levels of Undecidability in Rewriting. *Information and Computation*, 209(2):227–245, 2011.

3   J. Endrullis, J. Karhumäki, J.W. Klop, and A. Saarela. Degrees of infinite words, polynomials and atoms. In *Proc. Conf. Developments in Language Theory (DLT 2016)*, LNCS, pages 164–176. Springer, 2016.

4   J. Endrullis and J.W. Klop. De Bruijn's weak diamond property revisited. *Indagationes Mathematicae*, 24(4):1050–1072, 2013. In memory of N.G. (Dick) de Bruijn (1918–2012).

5   J. Endrullis, J.W. Klop, A. Saarela, and M. Whiteland. Degrees of transducibility. In *Proc. Conf. on Combinatorics on Words (WORDS 2015)*, volume 9304 of *LNCS*, pages 1–13. Springer, 2015.

───────────

[3]   Already the implication $DCR_1 \implies CP$ fails. To see this, consider the ARS $(2^{\mathbb{R}}, \rightarrow)$ where the steps are of the form $X \rightarrow X \cup \{y\}$ for $X \subseteq \mathbb{R}$ and $y \in \mathbb{R}$.

6   J.R. Hindley. *The Church–Rosser Property and a Result in Combinatory Logic.* PhD thesis, University of Newcastle-upon-Tyne, 1964.

7   G.P. Huet. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems. *Journal of the ACM*, 27(4):797–821, 1980.

8   J. Ketema and J.G. Simonsen. Least upper bounds on the size of confluence and church-rosser diagrams in term rewriting and λ-calculus. *ACM Trans. Comput. Log.*, 14(4):31:1–31:28, 2013.

9   J.W. Klop. *Combinatory Reduction Systems*, volume 127 of *Mathematical centre tracts.* Mathematisch Centrum, 1980.

10   J.W. Klop. Term Rewriting Systems. In *Handbook of Logic in Computer Science*, volume II, pages 1–116. Oxford University Press, 1992.

11   M. Li and P.M.B. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications.* Springer, 2nd edition, 2008.

12   M.H.A. Newman. On Theories with a Combinatorial Definition of "Equivalence". *Annals of Mathematics*, 42(2):223–243, 1942.

13   B.K. Rosen. Tree-manipulating systems and Church-Rosser theorems. *Journal of the ACM*, 20:160–187, 1973.

14   J.R. Shoenfield. *Degrees of Unsolvability.* North-Holland, Elsevier, 1971.

15   Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 2003.

16   R. Treinen. The first-order theory of one-step rewriting is undecidable. In *Proc. Conf. on Rewriting Techniques and Applications (RTA)*, volume 1103 of *LNCS*, pages 276–286. Springer, 1996.

17   V. van Oostrom. Confluence by Decreasing Diagrams. *Theoretical Computer Science*, 126(2):259–280, 1994.

18   V. van Oostrom. *Confluence for Abstract and Higher-Order Rewriting.* PhD thesis, Vrije Universiteit Amsterdam, 1994.

19   V. van Oostrom. Confluence by Decreasing Diagrams, Converted. In *Proc. Conf. on Rewriting Techniques and Applications (RTA 2008)*, volume 5117 of *LNCS*, pages 306–320. Springer, 2008.

20   H. Zantema. The Termination Hierarchy for Term Rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 12(1):3–19, 2001.

Coherence of Gray Categories via Rewriting

Simon Forest
LIX, École Polytechnique, France
simon.forest@lix.polytechnique.fr

Samuel Mimram
LIX, École Polytechnique, France
samuel.mimram@lix.polytechnique.fr

──── **Abstract** ────

Over the recent years, the theory of rewriting has been extended in order to provide systematic techniques to show coherence results for strict higher categories. Here, we investigate a further generalization to low-dimensional weak categories, and consider in details the first non-trivial case: presentations of tricategories. By a general result, those are equivalent to the stricter Gray categories, for which we introduce a notion of rewriting system, as well as associated tools: critical pairs, termination orders, etc. We show that a finite rewriting system admits a finite number of critical pairs and, as a variant of Newman's lemma in our context, that a convergent rewriting system is coherent, meaning that two parallel 3-cells are necessarily equal. This is illustrated on rewriting systems corresponding to various well-known structures in the context of Gray categories (monoids, adjunctions, Frobenius monoids). Finally, we discuss generalizations in arbitrary dimension.

The rewriting systems which are convergent have a fundamental property, which is a consequence of Newman's and other classical lemmas in rewriting theory: the space between any two rewriting zigzags with the same source and the same target can be filled with tiles witnessing the confluence of critical branchings. Otherwise said, every diagram commutes modulo the commutation of diagrams induced by critical branchings, which thus axiomatize the *coherence* of the structure.

Over the recent years, there have been many efforts to generalize the techniques of rewriting from words and terms to morphisms in strict n-categories, starting from the pioneering work of Burroni and Lafont [3, 15, 16]. Those widen the range of applicability of rewriting, and also allow a precise formulation of the above remark initially formulated by Squier, and generalized by Guiraud and Malbos by considering coherent presentations [17, 8, 9]. As a typical example, starting from the 2-category of planar binary forests, which is generated by a binary (μ) and a nullary corolla (η), one can consider rewriting rules expressing the fact that μ is associative and η is both a left and right unit for μ. The resulting rewriting system is convergent, and the technique described above allows to prove a coherence theorem for pseudomonoids, of which MacLane's coherence result is a particular case (a monoidal category is a pseudomonoid in the cartesian 2-category **Cat**).

It is of course of interest to generalize the coherence theorems for classical algebraic structures from strict to weak n-categories. For instance, coherence for pseudomonoids in tricategories is shown in [14]. A rewriting approach in this domain is desirable, but the way one could handle all the coherence morphisms present in weak categories was not

clear. Recently, the use of semistrict weak n-categories was advocated by the creators of the graphical proof-assistant Globular [1, 2] as a formalism adapted to computer manipulations: without loss of generality most of the coherence morphisms can be considered to be identities, excepting the interchangers and their coherences.

In this article, we develop the theory of rewriting for semistrict 3-categories, also called Gray categories, which is the first dimension in which the strict and weak definitions are not equivalent [6, 11]. We illustrate that this provides systematic principles for proving coherence of algebraic structures in Gray categories, allowing us to recover results in this direction recently proved [14, 4, 20] as well as new ones. Moreover, it turns out that this weak framework is better behaved than the strict one in some respects: it was observed that a finite rewriting system on strict 2-morphisms can give rise to an infinite number of critical branchings [15, 8], which is the source of many difficulties [16], both of theoretical and practical nature, whereas in the present setting we show that only a finite number of critical pairs can be generated. Finally, we also hint at generalizations in arbitrary dimension.

1   Coherent presentations of Gray categories

1.1   Sesquicategories

We begin by recalling the notion of 2-category as a variant of sesquicategories, details can be found in [18]. A *sesquicategory*, or *2-precategory*, C consists of

- a set C_0 of 0-*cells*,
- a set $C_1(x, y)$ of 1-*cells* $u : x \to y$ for every 0-cells x and y,
- a set $C_2(u, v)$ of 2-*cells* $\alpha : u \Rightarrow v : x \to y$ every parallel 1-cells $u, v : x \to y$,
- an identity 1-cell $1_x : x \to x$ for every 0-cell x,
- a composition function which to every 1-cells $u : x \to y$ and $v : y \to z$ associates a 1-cell $u * v : x \to y$,
- an identity 2-cell $1_u : u \Rightarrow u$ for every 1-cell u,
- a *vertical* composition function which to every 2-cell $\alpha : v \Rightarrow v'$ and $\beta : v' \Rightarrow v''$ associates a 2-cell $\alpha * \beta : v \Rightarrow v''$ (middle of (1)),
- a *left whiskering* composition function which to every 1-cell $u : x' \to x$ and 2-cell $\alpha : v \Rightarrow v'$ associates a 2-cell $u * \alpha : u * v \Rightarrow u * v' : x \to y'$ (left of (1)),
- a *right whiskering* composition function which to every 2-cell $\alpha : v \Rightarrow v'$ and 1-cell $w : y \to y'$ associates a 2-cell $\alpha * w : v * w \Rightarrow v' * w : x \to y'$ (right of (1)).

$$x' \xrightarrow{\ u\ } x \underset{v'}{\overset{v}{\rightrightarrows}}{\Downarrow\alpha}\, y \qquad\qquad x \underset{v''}{\overset{v}{\Rrightarrow}}\begin{smallmatrix}\Downarrow\alpha\\ \Downarrow\beta\end{smallmatrix} y \qquad\qquad x \underset{v'}{\overset{v}{\rightrightarrows}}{\Downarrow\alpha}\, y \xrightarrow{\ w\ } y' \tag{1}$$

such that compositions are associative and admit identities as neutral elements: for suitably typed 0-cells x, y, 1-cells u, v, w and 2-cells α, β, γ,

$$
\begin{array}{llll}
(u * v) * w = u * (v * w) & 1_x * u = u & & u * 1_y = u \\
(\alpha * \beta) * \gamma = \alpha * (\beta * \gamma) & 1_u * \alpha = \alpha & & \alpha * 1_v = \alpha \\
u * (\alpha * \beta) = (u * \alpha) * (u * \beta) & u * 1_v = 1_{u*v} & (\alpha * \beta) * w = (\alpha * w) * (\beta * w) & 1_v * w = 1_{v*w} \\
(u * v) * \alpha = u * (v * \alpha) & 1_x * \alpha = \alpha & \alpha * (v * w) = (\alpha * v) * w & \alpha * 1_y = \alpha \\
(u * \alpha) * w = u * (\alpha * w) & & &
\end{array}
\tag{2}
$$

In a composition, the dimension of the involved cells determines which composition is used, which allows us to unambiguously denote them by the same symbol. In a more terse way,

the category of sesquicategories can be defined as the category of categories enriched over **Cat** equipped with the "funny tensor product" [5].

A *2-category* C is a sesquicategory such that the interchange law holds: this means that for every 2-cells $\alpha : u \Rightarrow u' : x \to y$ and $\beta : v \Rightarrow v' : y \to z$, we have

$$(\alpha * v) * (u' * \beta) = (u * \beta) * (\alpha * v') \qquad x \xrightarrow[u']{\;\;u\;\Downarrow\alpha\;\;} y \xrightarrow[\Downarrow\beta]{\;v\;} z \;\; = \;\; x \xrightarrow[\Downarrow\alpha]{\;u\;} y \xrightarrow[v']{\;\;v\;\Downarrow\beta\;\;} z \quad (3)$$

Since both of the above compositions are equal, we can define the 0-composition of α and β to be either of them. By contrast, in a sesquicategory, the 0-composition of 2-cells does not make sense: we can only compose 2-cells in codimension 1.

1.2 Signatures

In the following, we will be interested in rewriting morphisms in freely generated sesquicategories. Recall that a *graph* consists of
- a set P_0 of vertices,
- a set P_1 of edges,
- functions $\mathsf{s}_0, \mathsf{t}_0 : \mathsf{P}_1 \to \mathsf{P}_0$ associating to each edge its source and target vertex.

We write P_1^* for the set of paths in the graph, $\mathsf{s}_0^*, \mathsf{t}_0^* : \mathsf{P}_1^* \to \mathsf{P}_0$ the source and target functions on paths, and uv for the concatenation of composable paths u and v.

A *signature* P consists of
- a graph $(\mathsf{P}_0, \mathsf{s}_0, \mathsf{t}_0, \mathsf{P}_1)$ whose vertices and edges are called 0- and 1-*generators*,
- a set P_2 of 2-*generators* together with functions $\mathsf{s}_1, \mathsf{t}_1 : \mathsf{P}_2 \to \mathsf{P}_1^*$ such that $\mathsf{s}_0^* \circ \mathsf{s}_1 = \mathsf{s}_0^* \circ \mathsf{t}_1$ and $\mathsf{t}_0^* \circ \mathsf{s}_1 = \mathsf{t}_0^* \circ \mathsf{t}_1$.

We write $a : x \to y$ to indicate that a is a 1-generator with $\mathsf{s}_0(a) = x$ and $\mathsf{t}_0(a) = y$, and similarly for 2-generators $\alpha : u \Rightarrow v$ with $\mathsf{s}_1(\alpha) = u$ and $\mathsf{t}_1(\alpha) = v$.

▶ **Example 1** (Monoids)**.** The signature for *monoids* is

$$\mathsf{P}_0 = \{\star\} \qquad \mathsf{P}_1 = \{1 : \star \to \star\} \qquad \mathsf{P}_2 = \{\mu : 2 \Rightarrow 1, \eta : 0 \Rightarrow 1\}$$

Note that the set P_1^* is isomorphic to $\mathbb{N}$, thus the notation for its elements. The 2-generators of this signature should respectively be understood as a formal multiplication (μ) and unit (η), which we will use below to express the structure of a monoid.

A signature P freely generates a sesquicategory with P_0 as 0-cells, P_1^* as 1-cells (composition being concatenation and identities empty paths), and whose 2-cells are generated by P_2. We write P_2^* for its set of 2-cells, whose elements can be described explicitly as follows.

▶ **Proposition 2.** *The 2-cells in* P_2^* *can be described as the sequences of the form*

$$(u_1 * \alpha_1 * w_1) * (u_2 * \alpha_2 * w_2) * \ldots * (u_n * \alpha_n * w_n)$$

with $u_i : x \to x_i$ *in* P_1^*, $\alpha_i : v_i \Rightarrow v_i' : x_i \to y_i$ *in* P_2, $w_i : y_i \to y$ *in* P_1^* *(the compositions above are formal ones). The canonical inclusion* $\mathsf{P}_2 \to \mathsf{P}_2^*$ *sends a 2-generator* $\alpha : u \Rightarrow v : x \to y$ *to* $(1_x * \alpha * 1_y)$, *vertical composition is given by concatenation, left whiskering the above morphism by* u *amounts to replace each* u_i *by* uu_i, *and similarly for right whiskering.*

Proof. The above sequences are the normal forms for a suitable orientation of the relations (2) as a convergent rewriting system on formal expressions. ◀

As customary, such morphisms can be pictured using string diagrams. For instance, in the signature of monoids (Ex. 1), if we draw μ by $\curlyvee$, we can picture the following morphisms:

$$(0 * \mu * 2) * (1 * \mu * 0) * \mu = \qquad\qquad (2 * \mu * 0) * (0 * \mu * 1) * \mu =$$

Note that in these pictures, there can be only one generator at a given height, and the relative heights matter, so that the two 2-cells are not considered to be equal (contrarily to 2-categories).

1.3 Rewriting systems

A *rewriting system* consists of a signature P together with a set P_3 of 3-*generators*, or *rewriting rules*, equipped with source and target functions $\mathsf{s}_2, \mathsf{t}_2 : \mathsf{P}_3 \to \mathsf{P}_2^*$. A *rewriting step*
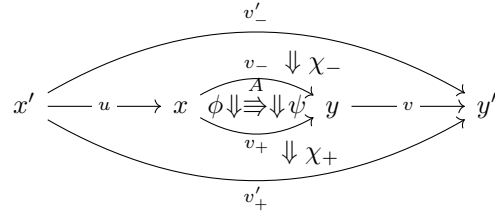
$$\chi_- * (u * A * v) * \chi_+ \quad : \quad \chi_- * (u * \phi * v) * \chi_+ \quad \Rightarrow \quad \chi_- * (u * \psi * v) * \chi_+$$

consists of a rewriting rule $A : \phi \Rightarrow \psi : v_- \Rightarrow v_+ : x \to y$ together with 1-cells $u : x' \to x$, $v : y \to y'$ and 2-cells $\chi_- : v_-' \Rightarrow v_-$, $\chi_+ : v_+ \Rightarrow v_+'$ as on the right above. A *rewriting path* is a finite sequence of composable rewriting steps $R_i : \phi_i \Rightarrow \psi_i$, with $\phi_{i+1} = \psi_i$.

▶ **Example 3.** The rewriting system for monoids has, on the signature of Ex. 1, the rules

$$A : (\mu * 1) * \mu \Rightarrow (1 * \mu) * \mu \qquad\qquad L : (\eta * 1) * \mu \Rightarrow \mu \qquad\qquad R : (1 * \eta) * \mu \Rightarrow \mu$$
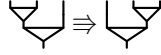
There is, for instance, a rewriting step

$$(3 * \mu * 1) * (1 * A * 1) * ((\mu * 1) * \mu) \quad : \qquad\qquad \Rightarrow$$

We write P_3^* for the set of rewriting paths, and $\mathsf{s}_2^*, \mathsf{t}_2^* : \mathsf{P}_3^* \to \mathsf{P}_2^*$ for the associated source and target functions. We can form a 3-precategory, noted P^*, with P_i^* as i-cells for $i = 0, 1, 2, 3$ (by convention $\mathsf{P}_0^* = \mathsf{P}_0$) and expected compositions. The notion of 3-precategory will be detailed in Sec. 4.1, but we can already say that it is the expected generalization of 2-precategories (see Sec. 1.1) in dimension 3: a *3-precategory* consists of a set C_i of i-cells for $i = 0, 1, 2, 3$ together with their source and target in lower dimension (except for 0-cells), identities for $0, 1, 2$-cells, and compositions between composable i- and j-cells, with $i, j = 1, 2, 3$, so that compositions are associative and unital in a suitable way. Note that, contrarily to 3-categories, there is only one kind of composition between i- and j-cells: those can only be composed in codimension $i \wedge j - 1$ (we write $i \wedge j$ for the minimum of i and j), which again allows to unambiguously use the same symbol for all compositions. A morphism of 3-precategories is called a *3-prefunctor*. By generalizing the argument of Prop. 2, one can show that P^* enjoys the following universal property, see Sec. 4.2 for details:

▶ **Proposition 4.** *The 3-precategory P^* is the free 3-precategory whose underlying 2-precategory is the one generated by the underlying signature of P and containing the rewriting rules as 3-cells.*

Following the terminology of [9], we say that two rewriting paths P and Q are *Peiffer-equivalent* when they differ only by successively permuting adjacent rewriting steps at disjoint positions, what we write $P \equiv Q$ below. For instance, with the notations of Ex. 3, the two following paths are Peiffer-equivalent:
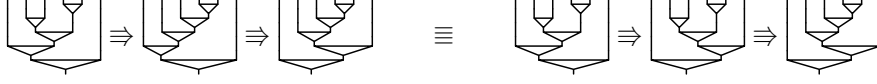
More generally, we can define the *Peiffer-equivalence* in a 3-precategory as the smallest congruence (w.r.t. compositions) such that, with cells as on the left, we have the relation on the right:

$$
\begin{array}{c}
((u_1 * A_1 * w_1) * \chi * (u_2 * \phi_2 * w_2)) \\
* \\
((u_1 * \psi_1 * w_1) * \chi * (u_2 * A_2 * w_2)) \\
\equiv \\
((u_1 * \phi_1 * w_1) * \chi * (u_2 * A_2 * w_2)) \\
* \\
((u_1 * A_1 * w_1) * \chi * (u_2 * \psi_2 * w_2))
\end{array}
\tag{4}
$$

1.4 (3,2)-precategories

A $(3,2)$-*precategory* is a 3-precategory in which every 3-cell $P : \phi \Rightarrow \psi$ is invertible: there exists a 3-cell $Q : \psi \Rightarrow \phi$ such that $P * Q = 1_\phi$ and $Q * P = 1_\psi$.

Given a rewriting system P, consider the rewriting system Q with $\mathsf{Q}_i = \mathsf{P}_i$ for $i = 0, 1, 2$ and $Q_3 = \mathsf{P}_3 \sqcup \mathsf{P}_3^-$ where $\mathsf{P}_3^- = \{A^- : \psi \Rightarrow \phi \mid A : \phi \Rightarrow \psi \in \mathsf{P}_3\}$ is the set of formally reverted rules in P_3. We write $P_3^\top$ for the set of 3-cells in Q_3^* quotiented by the smallest congruence such that $A * A^- = 1_\phi$ and $A^- * A = 1_\psi$ for every generator A in P_3, and call its elements *rewriting zigzags*. We can form a 3-precategory, noted $\mathsf{P}^\top$, with P_i^* as i-cells for $i = 0, 1, 2$, $\mathsf{P}_3^\top$ as 3-cells, and expected compositions: it is defined as P^* excepting that 3-cells are rewriting zigzags instead of rewriting paths.

▶ **Proposition 5.** *The* 3-*precategory* $\mathsf{P}^\top$ *is the free* $(3,2)$-*precategory on the* 3-*precategory* P^*.

According to the above proposition, we generalize the above notation and write P^- for the inverse of an arbitrary 3-cell P. Note that any 3-cell decomposes as $P_1^- * Q_1 * \ldots * P_n^- * Q_n$ where P_i and Q_i are rewriting paths (thus the terminology of rewriting zigzag). In the following, we will be mostly interested in $(3,2)$-precategories (as opposed to 3-precategories).

1.5 Presentations

In order to describe interesting 3-precategories using rewriting systems, we need to be able to quotient the 3-cells in the freely generated 3-precategory. A *presentation* consists of a rewriting system P equipped with a set P_4 of *relations* together with functions $\mathsf{s}_3, \mathsf{t}_3 : \mathsf{P}_4 \to \mathsf{P}_3^\top$ indicating the source and the target rewriting path of a relation, such that $\mathsf{s}_2^* \circ \mathsf{s}_3 = \mathsf{s}_2^* \circ \mathsf{t}_3$ and $\mathsf{t}_2^* \circ \mathsf{s}_3 = \mathsf{t}_2^* \circ \mathsf{t}_3$ (relations are between rewriting paths with same source and same target). We often write $\Gamma : P \Rrightarrow Q$ to indicate that Γ is a relation with $\mathsf{s}_3(\Gamma) = P$ and $\mathsf{t}_3(\Gamma) = Q$. We denote by $\equiv_{\mathsf{P}_4}$ (or sometimes even $\equiv$), the smallest congruence on 3-cells in $\mathsf{P}_3^\top$ such that $P \equiv_{\mathsf{P}_4} Q$ for every relation $\Gamma : P \Rrightarrow Q$ in P_4.

The $(3,2)$-precategory $\overline{\mathsf{P}}$ *presented* by a presentation P is the 3-category obtained from $\mathsf{P}^\top$ by quotienting 3-cells under $\equiv_{\mathsf{P}_4}$.

▶ Remark. A typical relation that one would like to express in the rewriting system of monoids (Ex. 3) is the fact that the two ways of multiplying the unit by itself are the same, as pictured below. However, in order to do so, we need to be able to "exchange" the two units (the first 3-cell on the right), which there is no way to achieve for now. This motivates looking at rewriting systems with more structure in next section.

$$\mathsf{Y}\!\mathsf{Q} \Rightarrow \mathsf{Q} \quad\equiv\quad \mathsf{Y}\!\mathsf{Q} \Rightarrow \mathsf{Q}\!\mathsf{J} \Rightarrow \mathsf{Q} \qquad\qquad \eta * R \quad\equiv\quad (X_{\eta,\eta} * \mu) * (\eta * L)$$

1.6 Presentations of Gray categories

We have seen above that a rewriting system freely generates a 3-precategory. In practice, we will be interested in describing 3-precategories having some additional structure and axioms.

A *Gray category* C is a 3-precategory equipped, for every pair of 2-cells ϕ and ψ as on the left, of an invertible 3-cell $X_{\phi,\psi}$ as on the right, called *interchanger*:

$$X_{\phi,\psi} : \qquad (\phi * v) * (u' * \psi) \Rrightarrow (u * \psi) * (\phi * v')$$

$$x \overset{u}{\underset{u'}{\Downarrow\phi}} y \overset{v}{\underset{v'}{\Downarrow\psi}} z \qquad\qquad x \xrightarrow[u']{u} y \underset{v'}{\overset{v}{\Downarrow\psi}} z \;\Rightarrow\; x \underset{u'}{\overset{u}{\Downarrow\phi}} y \overset{v}{\underset{v'}{\Downarrow\psi}} z \qquad (5)$$

such that

1. Peiffer-equivalences are identities,
2. interchangers are compatible with compositions and identities in all sensible ways: for example,

$$X_{\phi_1 * \phi_2, \psi} = ((\phi_1 * v) * X_{\phi_2, \psi}) * (X_{\phi_1, \psi} * (\phi_2 * v')) \qquad \text{and} \qquad X_{1_u, \psi} = 1_{u * \psi}$$

3. interchangers are natural: in the situation (5), given a 3-cell $P : \phi \Rrightarrow \phi'$

$$((P * v) * (u' * \psi)) * X_{\phi', \psi} = X_{\phi, \psi} * ((u * \psi) * (P * v'))$$

and symmetrically.

Alternatively, a Gray category can be defined to be category enriched over the category $\mathbf{Cat}_2$ of 2-categories equipped with a suitable tensor product, called the Gray tensor product [7]. A *Gray* $(3, 2)$-*category* is a Gray category in which every 3-cell is invertible. A *Gray functor* $f : C \to D$ between Gray categories is a 3-prefunctor preserving interchangers (we only consider the strict flavor of such functors here).

The notion of Gray category generalizes 3-categories by asking for explicit interchange cells: a 3-category is precisely a Gray category where all interchange 3-cells are identities. The relevance of Gray categories is that, although they are quite strict (compositions are strictly associative), they capture the full generality of *weak* 3-categories, as shown by the coherence theorem of Gordon, Power and Street [6, 11]:

▶ **Theorem 6.** *Every tricategory is (suitably) equivalent to a Gray category.*

In order to present Gray categories, we should ensure that our presentations generate interchangers and satisfy the required axioms. A *Gray presentation* P is a presentation such that

1. for every pair of 3-generators A_1 and A_2, as well as morphisms as on the left of (4), there is a relation as on the right of (4) called a *Peiffer generator*,

2. for every 2-generators α and β and 1-cell v as below:

$$x \underset{u'}{\overset{u}{\Downarrow\alpha}} x' \xrightarrow{\ v\ } y' \underset{w'}{\overset{w}{\Downarrow\beta}} z$$

left, there is a 3-generator $X_{\alpha,v,\beta}$, called *interchange generators*, as below:

$$X_{\alpha,v,\beta}: \qquad (\alpha * v * w) * (u' * v * \beta) \Rrightarrow (u * v * \beta) * (\alpha * v * w')$$

$$x \xrightarrow[u']{\overset{u}{\Downarrow\alpha}} x' \xrightarrow{\ v\ } y' \underset{w'}{\overset{w}{\Downarrow\beta}} z \ \Rrightarrow\ x \underset{u'}{\overset{u}{\Downarrow\alpha}} x' \xrightarrow{\ v\ } y' \xrightarrow[w']{\overset{w}{\Downarrow\beta}} z$$

and we write $\mathsf{P}_X \subseteq \mathsf{P}_3$ for the set of interchange generators,

3. for every 3-generator A, 1-cell v and 2-generator α as on the left or on the right below

$$x \underset{u'}{\overset{u}{\phi\Downarrow\overset{A}{\Rrightarrow}\Downarrow\psi}} x' \xrightarrow{\ v\ } y' \underset{w'}{\overset{w}{\Downarrow\alpha}} y \qquad\qquad x \underset{u'}{\overset{u}{\Downarrow\alpha}} x' \xrightarrow{\ v\ } y' \underset{w'}{\overset{w}{\phi\Downarrow\overset{A}{\Rrightarrow}\Downarrow\psi}} y \quad (6)$$

there is respectively a relation, called *interchange naturality generator*,

$$((A * v * w) * (u' * v * \alpha)) * X_{\psi,v*\alpha} \ \Rrightarrow\ X_{\phi,v*\alpha} * ((u * w * \alpha) * (A * v * w'))$$
$$((\alpha * v * w) * (u' * v * A)) * X_{\alpha*v,\psi} \ \Rrightarrow\ X_{\alpha*v,\phi} * ((u * v * A) * (\alpha * v * w'))$$

where the interchangers $X_{\alpha*v,\psi}$ are suitable composite of interchange generators (see proposition below).

The above families of 3- and 4-cells are called the *structural generators* of the presentation. We will not insist much about it in the following, but the choice of structural cells is implicitly supposed to be part of a Gray presentation.

▶ **Proposition 7.** *Given a Gray presentation* P*, the presented* $(3,2)$*-precategory* $\overline{\mathsf{P}}$ *is canonically a Gray* $(3,2)$*-category.*

Proof sketch. The first family of relations of P generates, by congruence, all the Peiffer equivalences, the second family of 3-cells generates, by composition, all the interchangers, and the third family of relations generates, by congruence, all the naturality conditions. ◀
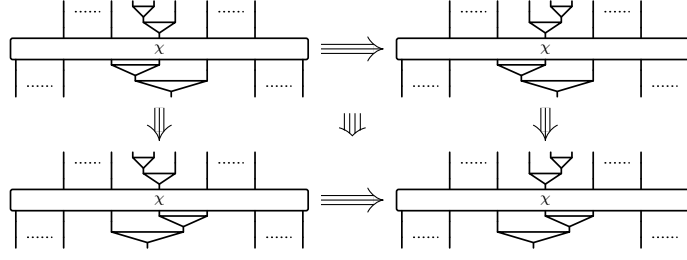
▶ **Example 8.** The Gray presentation of monoids consists of the rewriting system of Ex. 3, as well as additional interchange generators

together with the relations

(7)

as well as Peiffer generators, e.g.

for an arbitrary 2-cell $\chi : n + 1 \Rightarrow n + 3$, and interchange naturality generators, e.g.

(8)
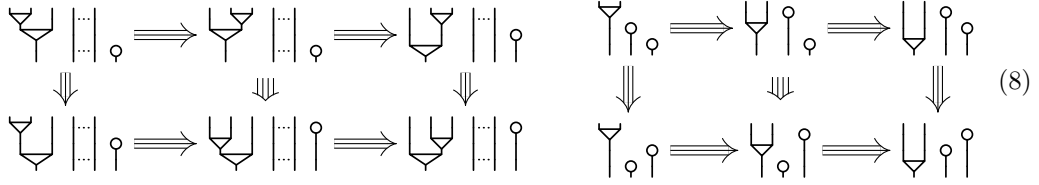
In the following, when describing a Gray presentation, we will not mention the structural cells which are always implicitly supposed to be present.

A *model* of a presentation P in a Gray category C is a Gray functor $\overline{\mathsf{P}} \to C$ from the presented Gray $(3,2)$-category to C.

▶ **Example 9.** A model of the presentation P of monoids (Ex. 8) in a Gray category C consists in a 1-cell $a : x \to x$ together with 2-cells $\mu : a * a \Rightarrow a$ and $\eta : 1_x \Rightarrow a$ and invertible 3-cells A, L, R (as in Ex. 3) satisfying suitable relations (as in Ex. 8). This is precisely what is usually called a *pseudomonoid* in C.

▶ Remark. A notion of presented Gray category (as opposed to $(3,2)$-category) can also be defined: it is slightly more involved since we still need to formally invert (by a localization) some morphisms, at least the interchangers. Similarly, we could consider their models which are functors to Gray categories. However, in practice people consider algebraic structures with invertible 3-cells (e.g. pseudomonoids), which explains why we are mostly interested in Gray $(3,2)$-categories here for simplicity.

Our goal is to show that some presentations are coherent, meaning that all the diagrams made of structural morphisms commute in the models. Formally, a Gray category is *coherent* when between any pair of parallel 2-cells there is at most one 3-cell and a Gray presentation is *coherent* when the associated Gray $(3,2)$-category is.

2 Rewriting

2.1 Confluence

Every rewriting system induces an abstract rewriting system (i.e., a graph) with 2-cells in P_2^* as vertices and rewriting steps as edges (the set of paths thus being P_3^*), from which we can use the classical notions and properties of rewriting theory, detailed below. We slightly depart from the tradition by, for confluence properties, asking that diagrams should be closed and commute <u>modulo the relations</u> in P_4.

Given a rewriting path $P : \phi \Rightarrow \psi$, we say that ϕ *rewrites* to ψ. A *normal form* is a 2-cell ϕ such that the only rewriting path with source ϕ is the empty one. A *branching* is

a pair of coinitial rewriting paths $P_1 : \phi \Rightarrow \phi_1$ and $P_2 : \phi \Rightarrow \phi_2$; it is *local* when both P_1 and P_2 are rewriting steps, it is *joinable* when there exists a pair of cofinal rewriting paths $Q_1 : \phi_1 \Rightarrow \psi$ and $Q_2 : \phi_2 \Rightarrow \psi$, it is *confluent* when there exists a pair of cofinal rewriting paths $Q_1 : \phi_1 \Rightarrow \psi$ and $Q_2 : \phi_2 \Rightarrow \psi$ such that $P_1 * Q_1 \equiv_{\mathsf{P}_4} P_2 * Q_2$, see left of (9). Similarly, a rewriting zigzag $P : \phi_1 \Rightarrow \phi_2$ in $\mathsf{P}_3^\top$ is *confluent* when there exists a pair of cofinal rewriting paths $Q_1 : \phi_1 \Rightarrow \psi$ and $Q_2 : \phi_2 \Rightarrow \psi$ such that $P * Q_2 \equiv Q_1$, see right of (9)

$$
\begin{array}{cc}
\begin{array}{c}
\phi \\
P_1 \swarrow \quad \searrow P_2 \\
\phi_1 \quad \equiv \quad \phi_2 \\
Q_1 \searrow \quad \swarrow Q_2 \\
\psi
\end{array}
&
\begin{array}{c}
P \\
\phi_1 \quad \equiv \quad \phi_2 \\
Q_1 \searrow \quad \swarrow Q_2 \\
\psi
\end{array}
\end{array}
\tag{9}
$$

A rewriting system is

- *terminating* when every sequence of composable rewriting steps is finite,
- (*locally*) *confluent* when every (local) branching is confluent,
- *Church-Rosser* when every rewriting zigzag is confluent,
- *convergent* if both terminating and locally confluent.

In a terminating rewriting system, every 2-cell ϕ rewrites to a normal form $\hat\phi$. The classical proof by well-founded induction of Newman's lemma [19], can be directly adapted (as in [8, Thm. 3.1.6]) in order to show

▶ **Theorem 10.** *A convergent rewriting system is confluent.*

Finally, for abstract rewriting systems it is well known that confluence implies the Church-Rosser property. In this setting, this translates as the following theorem, which adapts in our setting, the proof of Squier's theorem for coherent presentations of categories, see [17, Thm. 5.2] and [8, Thm. 4.3.2]:

▶ **Theorem 11.** *A convergent presentation* P *is Church-Rosser and coherent.*

Proof. Suppose given a rewriting path $P : \phi \Rightarrow \psi$. Since P is terminating, there is a rewriting path $P_\phi : \phi \Rightarrow \hat\phi$ (resp. $P_\psi : \psi \Rightarrow \hat\psi$) from ϕ (resp. ψ) to a normal form $\hat\phi$ (resp. $\hat\psi$). Moreover, by confluence, we have $\hat\phi = \hat\psi$ and $P_\phi \equiv P * P_\psi$, see the left of (10). Therefore, we have equivalences $P \equiv P_\phi * P_\psi^-$ and $P^- \equiv P_\psi * P_\phi^-$, as in the middle and right of (10):

$$
\begin{array}{ccc}
\begin{array}{c}
\phi \overset{P}{\Longrightarrow} \psi \\
P_\phi \searrow \equiv \swarrow P_\psi \\
\hat\phi
\end{array}
&
\begin{array}{c}
\phi \overset{P}{\Longrightarrow} \psi \\
P_\phi \searrow \equiv \swarrow P_\psi^- \\
\hat\phi
\end{array}
&
\begin{array}{c}
\phi \overset{P^-}{\Longleftarrow} \psi \\
P_\phi^- \nearrow \equiv \nwarrow P_\psi \\
\hat\phi
\end{array}
\end{array}
\tag{10}
$$

Finally, as explained above, a 3-cell of $\overline{\mathsf{P}}$ is a zigzag of rewriting paths $P_1^- * Q_1 * \ldots * P_n^- * Q_n$ which is equivalent (modulo relations and axioms for inverses) to $P_\phi * P_\psi^-$:

$$
\begin{array}{c}
\phi \overset{P_1^-}{\Longrightarrow} \psi_1 \overset{Q_1}{\Longrightarrow} \ldots \overset{P_n^-}{\Longrightarrow} \psi_n \overset{Q_n}{\Longrightarrow} \psi \\
P_\phi \Downarrow \quad \equiv \quad P_{\psi_1}^- = P_{\psi_1} \equiv \quad \equiv \quad = P_{\psi_n} \equiv \quad \Downarrow P_\psi^- \\
\hat\phi = = = = \hat\phi = = \ldots = = \hat\phi = = = = \hat\phi
\end{array}
$$

Note that the 3-cell $P_\phi * P_\psi^-$ only depends on the source ϕ and the target ψ. We immediately deduce that two parallel 3-cells in $\overline{\mathsf{P}}$ are equal.　◀

2.2   Termination

Termination of a presentation is usually proved by checking that rules are decreasing according to some suitable order. A *termination order* is a well-founded partial order $<$ on parallel 2-cells of a presentation P such that

- for every rewriting rule $A : \phi \Rightarrow \psi$ we have $\phi > \psi$,
- given composable 2-cells ϕ, ψ_1 and ϕ' (resp. ϕ, ψ_2 and ϕ') such that $\psi_1 > \psi_2$, we have $\phi * \psi_1 * \phi' > \phi * \psi_2 * \phi'$
- given 2-cells $\phi > \psi$ and composable 1-cells u and w, we have $u * \phi * w > u * \psi * w$.

▶ **Proposition 12.** *A rewriting system equipped with a termination order is terminating.*

▶ **Example 13.** A termination order for the rewriting system of monoids (Ex. 3) can be constructed as follows. Firstly, the three non-structural rewriting rules can be shown to be terminating exactly as for 3-polygraph of monoids [15, Sect. A.2] (roughly L and R decrease the number of generators and A puts μ generators on the right), by a termination order for which the interchangers are left invariant. Secondly, the interchangers make 2-cells decrease in the following sense. A 2-cell corresponds to a forest of leveled planar binary trees (where nodes correspond to 2-generators), i.e., trees equipped with a total "vertical" order refining the depth order. The interchanger rules decrease the sum, for each generators, of the number of generators above (w.r.t. to the vertical order) and on the left (which is easily defined for such forests).

2.3   Critical branchings

Given a local branching (P_1, P_2), the following situations can occur. The branching is
- *trivial* when $P_1 = P_2$,
- *non-minimal* when there is another branching (Q_1, Q_2) such that $P_i = \phi * (u * Q_i * v) * \psi$ for $i = 0, 1$ for some 1-cells u, v and 2-cells ϕ, ψ, not all identities,
- *independent*, or *Peiffer*, when there are morphisms of the form (4) such that

$$P_1 = ((u_1 * A_1 * w_1) * \chi * (u_2 * \phi_2 * w_2)) \qquad P_2 = ((u_1 * \phi_1 * w_1) * \chi * (u_2 * A_2 * w_2))$$

- *natural* when there are morphisms as on the left of (6) such that

$$P_1 = ((A * v * w) * (u' * v * \alpha))$$

P_2 is the first rewriting step of $X_{\phi, v*\alpha}$, and similarly for the situation on the right of (6),
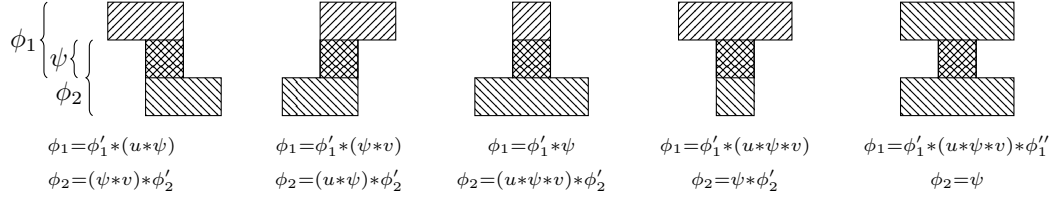- *critical* when it is of none of the above forms.

Since, by definition of Gray presentations, non-critical branchings are necessarily confluent, we have:

▶ **Theorem 14.** *A presentation is locally confluent if and only if every critical branching is confluent.*

As usual, critical branchings can be computed by considering the ways two left members ϕ_1 and ϕ_2 of rules can overlap non-trivially (sharing at least one 2-generator). Graphically, the following generic situations can happen, where the two regions respectively represent ϕ_1 and ϕ_2, the square ψ in the middle being the intersection (overlap) of both, which is supposed

not to be an identity 2-cell:

$$\phi_1=\phi_1'*(u*\psi)$$
$$\phi_2=(\psi*v)*\phi_2'$$

$$\phi_1=\phi_1'*(\psi*v)$$
$$\phi_2=(u*\psi)*\phi_2'$$

$$\phi_1=\phi_1'*\psi$$
$$\phi_2=(u*\psi*v)*\phi_2'$$

$$\phi_1=\phi_1'*(u*\psi*v)$$
$$\phi_2=\psi*\phi_2'$$

$$\phi_1=\phi_1'*(u*\psi*v)*\phi_1''$$
$$\phi_2=\psi$$

(and also the situations obtained by swapping ϕ_1 and ϕ_2). From this, one deduces that any pair of rules can give rise to a finite number of critical branchings which can effectively be computed (the algorithmic aspects will be detailed in future works). Moreover, note that a non-structural rewriting rule $R : \phi \Rightarrow \psi$ can only give rise to a finite number of critical branchings with interchangers: if the two 2-generators involved in an interchanger $X_{\alpha,v,\beta}$ are too far apart horizontally (i.e., v is a composite of too many 1-cells), the branching is necessarily an exchange branching, e.g. left of (8). Similarly, that two interchangers never make a critical branching (all such branchings are natural), e.g. right of (8). From the above considerations, we deduce:

▶ **Theorem 15.** *A presentation with a finite number of 2-generators and of non-structural 3-generators, with non-identity 2-cells as sources, has a finite number of critical branchings.*

It should be noted that this theorem contrasts with the situation for presentations of $(3, 2)$-categories (where interchangers are identities), where a finite presentation can give rise to an infinite number of critical branchings [15, 8]. Our formalization of rewriting systems avoids this problem, at the cost of having to explicitly handle interchangers.

▶ **Example 16.** The presentation for monoids (Ex. 8) has five critical branchings:
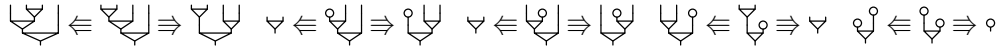
2.4 A coherent completion procedure

The general methodology for constructing confluent presentations is the following one. Suppose given a presentation P (usually containing no relation in P_4 excepting structural ones).

1. Find a termination order for the rules of P: if none can be found try to reorient some rules. Conclude that P is terminating by Prop. 12.
2. Compute the critical branchings and check that they are joinable: if a critical branching is not joinable, add a new rule to make it confluent (this is the Knuth-Bendix completion procedure [13]).
3. For every critical branching, choose a way to join it and add a corresponding relation in P_4 (if not already present).
4. Conclude that P is locally confluent by Thm. 14, thus confluent by Thm. 10 and thus coherent by Thm. 11.
5. Optionally, remove some redundant rules and relations in order to achieve a smaller presentation.

This methodology is illustrated in next section. Note that steps 2 and 3 can be combined, giving rise to a "homotopical completion procedure" and 5 can be partly automated: this is detailed in the case of coherent presentations of monoids in [10] and left for future work for Gray presentations.

3   Applications

3.1   Pseudomonoids

Consider the presentation P for monoids given in Ex. 8 whose termination was shown in Ex. 13. There are five critical branchings, given in Ex. 16, which are all joinable. If we add five corresponding relations in P we obtain a convergent, and thus coherent, presentation. Note however that the presentation P given in Ex. 8 has only two relations: in fact, three of the five relations are derivable from the other and can thus be removed (the argument given in [8] for pseudomonoids in 3-categories can directly be adapted to our setting). This allows us to recover the coherence theorem of [14].

3.2   Adjunctions

The presentation for *adjunctions* is given by $\mathsf{P}_0 = \{x, y\}$, $\mathsf{P}_1 = \{a : x \to y, b : y \to x\}$ and $\mathsf{P}_2 = \{\eta : 1_x \Rightarrow ab, \varepsilon : ba \Rightarrow 1_y\}$ where η and ε are respectively pictured as $\cap$ and $\cup$. The two rules are shown on the left below and the relations corresponding to the two critical branchings are on the right:

$$(11)$$

They are sometimes called the *swallowtail relations*. A model for this presentation in the 2-category **Cat** (seen as a $(3, 2)$-precategory with only identity 3-cells) is precisely an adjunction. Termination can be shown by observing that the two non-structural rules decrease the number of generators and the structural rules decrease the number of generators which are "on the left and above", as in the previous case. We deduce that this presentation is coherent, thus recovering a variant of the coherence theorem shown in [4] (see below).

3.3   Self-dualities

The theory for *self-dualities* is the following variant of the previous one. We have $\mathsf{P}_0 = \{\star\}$, $\mathsf{P}_1 = \{a : \star \to \star\}$, $\mathsf{P}_2 = \{\eta : 1_\star \Rightarrow aa, \varepsilon : aa \Rightarrow 1_\star\}$ where η and ε are respectively pictured as $\cap$ and $\cup$. The two rules are those on the left of (11). Note that because of the difference in "typing" of 0- and 1-cells, the rewriting system is not anymore terminating, since we have the reduction

Moreover, this endomorphism 3-cell is not an identity, preventing any hope for the presentation to be coherent. Following [4], we can still aim at showing a partial coherence result by restricting to 2-cells which are *connected*, i.e., whose graphical representation is connected (we do not give the formal definition here). In this case, termination can actually be shown by using the same arguments as in Sec. 3.2. However, the critical pairs are not joinable either since, for instance, we have

(for which there is little hope that a Knuth-Bendix completion will provide a reasonably small presentation). However, one can obtain a rewriting system which is terminating on connected 2-cells and confluent by orienting the interchangers as follows

$$\cup \, \big|\cdot\big| \, \cup \Rightarrow \cup \, \big|\cdot\big| \, \cup \qquad \cap \, \big|\cdot\big| \, \cap \Rightarrow \cap \, \big|\cdot\big| \, \cap \qquad \cap \, \big|\cdot\big| \, \cup \Rightarrow \cap \, \big|\cdot\big| \, \cup \qquad \cup \, \big|\cdot\big| \, \cap \Rightarrow \cup \, \big|\cdot\big| \, \cap$$

The relations generated by critical branchings can be pictured as on the right of (11).

3.4 Frobenius monoids

The presentation for (non-unital) *Frobenius monoids* is given by $\mathsf{P}_0 = \{\star\}$, $\mathsf{P}_1 = \{1 : \star \to \star\}$ and $\mathsf{P}_2 = \{\mu : 2 \Rightarrow 1, \delta : 1 \Rightarrow 2\}$. If we respectively picture μ and δ by $\curlyvee$ and $\curlywedge$, we have the four rewriting rules on the left below:

$$\curlywedge\!\!\curlyvee \Rightarrow \bowtie \qquad \curlyvee\!\!\curlywedge \Rightarrow \bowtie \qquad \curlyvee\!\!\curlyvee \Rightarrow \curlyvee\!\!\curlyvee \qquad \curlywedge\!\!\curlywedge \Rightarrow \curlywedge\!\!\curlywedge \qquad \ominus\!\!\curlyvee \Rightarrow \bowtie \qquad \ominus\!\!\curlywedge \Rightarrow \bowtie$$

(and interchangers are oriented as usual). By Knuth-Bendix completion, we add the two rules on the right. The resulting rewriting system has 19 joinable critical pairs, to each of which corresponds a relation. We conjecture that the rewriting system is terminating, which would give rise to a coherence theorem for Frobenius monoids. A coherence theorem using a different set of generators and relations is shown in [4].

4 Rewriting systems in higher dimension

4.1 Precategories

Given $n \in \mathbb{N}$, an *n-globular set* C is a diagram of sets

$$C_0 \overset{s_0}{\underset{t_0}{\Leftarrow}} C_1 \overset{s_1}{\underset{t_1}{\Leftarrow}} C_2 \overset{s_2}{\underset{t_2}{\Leftarrow}} \ldots \overset{s_{n-1}}{\underset{t_{n-1}}{\Leftarrow}} C_n$$

such that $s_i \circ s_{i+1} = s_i \circ t_{i+1}$ and $t_i \circ s_{i+1} = t_i \circ t_{i+1}$ for $0 \leq i < n-1$. A morphism $f : C \to D$ between n-globular sets is a family of morphisms $f_i : C_i \to D_i$, with $0 \leq i \leq n$, such that $s_i \circ f_{i+1} = f_i \circ s_i$. The resulting category is denoted by $\mathbf{Glob}_n$. Given $i, j, k \in \mathbb{N}$ with $k < i$ and $k < j$, we write $G_i \times_k G_j$ for the pullback of the diagram $C_i \overset{t_k \circ \ldots \circ t_{i-1}}{\longrightarrow} C_k \overset{s_k \circ \ldots \circ s_{j-1}}{\longleftarrow} C_j$.

An *n-precategory* C, see [12], is an n-globular set equipped with

- identity functions $1_i : G_i \to G_{i+1}$ for $0 \leq i < n$,
- composition functions $*_{i,j} : G_i \times_{i \wedge j - 1} G_j \to G_{i \vee j}$ for $0 < i, j \leq n$.

As previously, since the dimension of cells determines the functions to be used, we omit the indices from s, t, 1 and $*$. For composition, it is sometimes useful to write $u *_k v$ to indicate that $k = i \wedge j - 1$, where i is the dimension of u and j is the dimension of v. We require the following axioms:

- for $(u, v) \in C_i \times_{i \wedge j - 1} C_j$ with $0 < i, j \leq n$,

$$s(u * v) = \begin{cases} u * s(v) & \text{if } i < j \\ s(u) & \text{if } i = j \\ s(u) * v & \text{if } i > j \end{cases} \qquad t(u * v) = \begin{cases} u * t(v) & \text{if } i < j \\ t(v) & \text{if } i = j \\ t(u) * v & \text{if } i > j \end{cases}$$

- for every $u \in C_i$ with $0 \leq i < n$, $s(1_u) = u = t(1_u)$

▬ for every $(u, v) \in C_i \times_{i \wedge j - 1} C_j$ with $0 < i, j \leq n$,

$$1_u * v = \begin{cases} v & \text{if } i \leq j \\ 1_{u*v} & \text{if } i > j \end{cases} \qquad\qquad u * 1_v = \begin{cases} u & \text{if } i \geq j \\ 1_{u*v} & \text{if } i < j \end{cases}$$

such that, for composable cells u, v, w, with $k < l$,

$$(u *_k v) *_k w = u *_k (v *_k w) \qquad\qquad u *_k (v *_l w) = (u *_k v) *_l (u *_k w)$$
$$(u *_l v) *_k w = (u *_k v) *_l (u *_k w)$$

A morphism of n-precategories, called an n-*prefunctor*, is a morphism between the underlying globular sets which preserves identities and compositions as expected. We write $\mathbf{PCat}_n$ for the category of n-precategories. This category is locally presentable and thus complete and cocomplete. Given an n-precategory C, we write C_0 for its set of 0-cells seen as an n-category with empty sets of i-cells for $0 < i \leq n$. The "funny tensor product" $C \boxtimes D$ of two n-precategories C and D is defined as the pushout on the right where the arrows are the obvious inclusions. This makes $\mathbf{PCat}_n$ into a monoidal category and we have:

$$\begin{array}{ccc} C_0 \times D_0 & \longrightarrow & C \times D_0 \\ \downarrow & & \vdots \\ C_0 \times D & \cdots\cdots\!\!\longrightarrow & C \boxtimes D \end{array}$$

▶ **Proposition 17.** *An $n+1$-precategory is the same as a category enriched in $\mathbf{PCat}_n$ equipped with the funny tensor product.*

4.2 Prepolygraphs

We now briefly introduce the notion of *prepolygraph* which generalizes in arbitrary dimension the notion of rewriting system, by a direct adaptation the definition invented by Burroni for n-categories [3]. We write $\mathbf{PCat}_n^+$ for the pullback on the right where the arrow on the

$$\begin{array}{ccc} \mathbf{PCat}_n^+ & \cdots\!\!\longrightarrow & \mathbf{Glob}_{n+1} \\ \vdots & & \downarrow \\ \mathbf{PCat}_n & \longrightarrow & \mathbf{Glob}_n \end{array}$$

top is the forgetful functor and the one on the left is the truncation functor (forgetting the set of $n+1$-cells in an $n+1$-globular set). An object in this category consists of an n-precategory equipped with a set of $n+1$-cells (for which there is no notion of composition). There is a forgetful functor $\mathbf{PCat}_{n+1} \to \mathbf{PCat}_n^+$ which amounts to forget about compositions involving $n+1$-cells, which admits a right adjoint $L_n : \mathbf{PCat}_n^+ \to \mathbf{PCat}_{n+1}$, generating all the formal compositions of $n+1$-cells.

We now define by induction on $n \in \mathbb{N}$, the category $\mathbf{Pol}_n$ of n-*prepolygraphs* together with a functor $F_n : \mathbf{Pol}_n \to \mathbf{PCat}_n$ associating to each n-prepolygraph the associated freely generated n-precategory. For $n = 0$, we set $\mathbf{Pol}_0 = \mathbf{Set}$ and F_0 is the identity

$$\begin{array}{ccc} \mathbf{Pol}_{n+1} & \xrightarrow{\;F_n^+\;} & \mathbf{PCat}_n^+ \\ \downarrow & & \downarrow \\ \mathbf{Pol}_n & \xrightarrow[F_n]{} & \mathbf{PCat}_n \end{array}$$

functor ($\mathbf{PCat}_0$ is isomorphic to $\mathbf{Set}$). The category of $n+1$-prepolygraphs is defined by the pullback on the right where the vertical arrow is the expected forgetful functor, and we define the functor $F_{n+1} = L_{n+1} \circ F_n^+$. More explicitly, an n-prepolygraph consists in a diagram of sets

such that $\mathsf{s}_i^* \circ \mathsf{s}_{i+1} = \mathsf{s}_i^* \circ \mathsf{t}_{i+1}$ and $\mathsf{t}_i^* \circ \mathsf{s}_{i+1} = \mathsf{t}_i^* \circ \mathsf{t}_{i+1}$, together with a structure of n-precategory on the globular set on the bottom row: P_i is the set of i-*generators*, $\mathsf{s}_i, \mathsf{t}_i : \mathsf{P}_{i+1} \to \mathsf{P}_i^*$

respectively associate to each $i+1$-generator its *source* and *target*, and P_i^* is the set of *i-cells*, i.e., formal compositions of *i*-generators.

The cells in such prepolygraphs are particularly easy to manipulate because of the following normal form, generalizing Prop. 2 and its proof. We plan to investigate algorithmic aspects (for computing critical pairs, etc.) based on this representation in future works.

▶ **Theorem 18.** *A non-identity k-cell P in an n-prepolygraph decomposes uniquely as $P = R^1 * R^2 * \ldots * R^p$ with each R^i being a k-rewriting step, i.e., a composite of the form $R^i = u_{k-1}^i * (\ldots * (u_2^i * (u_1^i * A^i * w_1^i) * w_2^i) * \ldots) * w_{k-1}^i$ where A^i is a k-generator and u_j^i and v_j^i are j-cells.*

Interestingly, this formalization based on prepolygraphs corresponds precisely to the one proposed by Bar and Vicary [2]: their representation is more economical thanks to the use of integers in order to encode cells, but somewhat obscures the universal properties it satisfies.

▶ **Proposition 19.** *The n-signatures of [2] correspond to the n-prepolygraphs defined above.*

Their work gives hints at a way to generalize Gray presentations in order to present semistrict tetracategories, by providing the adapted collections of structural cells. We plan to investigate this, as well as an adaptation of our techniques in order to provide automation to their tool Globular [1] in future work.

References

1 Krzysztof Bar, Aleks Kissinger, and Jamie Vicary. Globular: an online proof assistant for higher-dimensional rewriting. In *LIPIcs*, volume 52, pages 34:1–34:11, 2016. `arXiv:1612.01093`.

2 Krzysztof Bar and Jamie Vicary. Data structures for quasistrict higher categories. In *Logic in Computer Science (LICS), 32nd Annual Symposium on*, pages 1–12. IEEE, 2017. `arXiv:1610.06908`.

3 Albert Burroni. Higher-dimensional word problems with applications to equational logic. *Theoretical computer science*, 115(1):43–62, 1993.

4 Lawrence Dunn and Jamie Vicary. Coherence for frobenius pseudomonoids and the geometry of linear proofs. Preprint, 2016. `arXiv:1601.05372`.

5 François Foltz, Christian Lair, and GM Kelly. Algebraic categories with few monoidal biclosed structures or none. *Journal of Pure and Applied Algebra*, 17(2):171–177, 1980.

6 Robert Gordon, Anthony John Power, and Ross Street. *Coherence for tricategories*, volume 558. American Mathematical Soc., 1995.

7 John Walker Gray. *Formal category theory: adjointness for 2-categories*, volume 391. Springer, 2006.

8 Yves Guiraud and Philippe Malbos. Higher-dimensional categories with finite derivation type. *Theory and Applications of Categories*, 22(18):420–478, 2009.

9 Yves Guiraud and Philippe Malbos. Polygraphs of finite derivation type. *Mathematical Structures in Computer Science*, pages 1–47, 2016. `arXiv:1402.2587`.

10 Yves Guiraud, Philippe Malbos, and Samuel Mimram. A homotopical completion procedure with applications to coherence of monoids. In *RTA-24th International Conference on Rewriting Techniques and Applications*, volume 21, pages 223–238, 2013.

11 Nick Gurski. *Coherence in three-dimensional category theory*, volume 201. Cambridge Univ. Press, 2013.

12 Aleks Kissinger and Jamie Vicary. Semistrict *n*-categories via rewriting. Proceedings of the first workshop on *Higher-Dimensional Rewriting and Applications*, 2015.

13   Donald E Knuth and Peter B Bendix. Simple word problems in universal algebras. In *Computational problems in abstract algebra*, pages 263–297, 1970.

14   Stephen Lack. A coherent approach to pseudomonads. *Advances in Math.*, 152(2):179–202, 2000.

15   Yves Lafont. Towards an algebraic theory of boolean circuits. *Journal of Pure and Applied Algebra*, 184(2):257–310, 2003.

16   Samuel Mimram. Towards 3-Dimensional Rewriting Theory. *Logical Methods in Computer Science*, 10(1):1–47, 2014. `arXiv:1403.4094`.

17   Craig C Squier, Friedrich Otto, and Yuji Kobayashi. A finiteness condition for rewriting systems. *Theoretical Computer Science*, 131(2):271–294, 1994.

18   Ross Street. Categorical structures. *Handbook of algebra*, 1:529–577, 1996.

19   Terese. *Term Rewriting Systems*. Number 55 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003.

20   Dominic Verdon. Coherence for braided and symmetric pseudomonoids. Preprint, 2017. `arXiv:1705.09354`.

Completeness of Tree Automata Completion

Thomas Genet

Univ Rennes/Inria/IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
genet@irisa.fr

──── **Abstract** ────

We consider rewriting of a regular language with a left-linear term rewriting system. We show a completeness theorem on equational tree automata completion stating that, if there exists a regular over-approximation of the set of reachable terms, then equational completion can compute it (or safely under-approximate it). A nice corollary of this theorem is that, if the set of reachable terms is regular, then equational completion can also compute it. This was known to be true for some term rewriting system classes preserving regularity, but was still an open question in the general case. The proof is not constructive because it depends on the regularity of the set of reachable terms, which is undecidable. To carry out those proofs we generalize and improve two results of completion: the Termination and the Upper-Bound theorems. Those theoretical results provide an algorithmic way to safely explore regular approximations with completion. This has been implemented in Timbuk and used to verify safety properties, automatically and efficiently, on first-order and higher-order functional programs.

1 Introduction

Given a term rewriting system (TRS for short) $\mathcal{R}$ and a tree automaton $\mathcal{A}$ recognizing a regular tree language $\mathcal{L}(\mathcal{A})$, the set of reachable terms is $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) = \{t \mid s \in \mathcal{L}(\mathcal{A}) \text{ and } s \rightarrow_{\mathcal{R}}^* t\}$. In this paper, we show that the *equational tree automata completion algorithm* [15] is complete w.r.t. regular approximations. If $\mathcal{R}$ is left-linear and there exists a regular language $\mathscr{L}$ over-approximating $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$, i.e., $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \mathscr{L}$ then completion can build a tree automaton $\mathcal{A}^*$ such that $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\mathcal{A}^*) \subseteq \mathscr{L}$. We also show that completion is complete w.r.t. TRSs preserving regularity, i.e., if $\mathscr{L} = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ then completion can build a tree automaton $\mathcal{A}^*$ such that $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) = \mathcal{L}(\mathcal{A}^*) = \mathscr{L}$. On the one hand, automata built by completion-like algorithms are known to recognize *exactly* the set of reachable terms, for some *restricted* classes of TRSs [17, 24, 8, 10]. On the other hand, automata completion is able to build *over-approximations* for *any* left-linear TRS [9, 23, 15], and even for non-left-linear TRSs [3]. Such approximations are used for program verification [5, 4, 10, 14] as well as to automate termination proofs [16, 20]. To define approximations, completion uses an additional set of equations E and builds a tree automaton $\mathcal{A}_{\mathcal{R},E}^*$ such that $\mathcal{L}(\mathcal{A}_{\mathcal{R},E}^*) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$. Starting from $\mathcal{R}$, $\mathcal{A}$, and E Timbuk[12] is an automatic tool to build $\mathcal{A}_{\mathcal{R},E}^*$. Until now it was an open question whether completion can build *any* regular over-approximation or compute the set of reachable terms if this set is regular. The *first contribution* of this paper is to answer these two questions in the positive, for general left-linear TRSs. The proofs are not

constructive but, the *second contribution* is to provide an efficient method to explore regular approximations for TRSs encoding functional programs.

For the approximated case, the proof of completeness is organized as follows. If there exists a regular over-approximation $\mathscr{L}$ such that $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \mathscr{L}$, we know that there exists a tree automaton $\mathcal{B}$ such that $\mathcal{L}(\mathcal{B}) = \mathscr{L}$. From $\mathcal{B}$, using the Myhill-Nerode theorem, we can infer a set of equations E such that the set of E-equivalence classes $\mathcal{T}(\mathcal{F})/_{=_E}$ is finite. Then we prove the following theorems:

(a) If $\mathcal{T}(\mathcal{F})/_{=_E}$ is finite, then it is possible to build from E a set of equations E', equivalent to E, such that completion of any automaton $\mathcal{A}$ by any TRS $\mathcal{R}$ with E' always terminates. This generalizes the termination theorem of [10];

(b) If $\mathcal{T}(\mathcal{F})/_{=_E}$ is finite, then it is possible to build from E and $\mathcal{A}$ a tree automaton $\mathbb{A}$ recognizing the same language as $\mathcal{A}$ such that the completed automaton $\mathbb{A}^*_{\mathcal{R},E}$ has the following precision property: $\mathcal{L}(\mathbb{A}^*_{\mathcal{R},E}) \subseteq \mathcal{R}^*_E(\mathcal{L}(\mathcal{A}))$, where $\mathcal{R}^*_E(\mathcal{L}(\mathcal{A}))$ is the set of reachable terms by rewriting modulo E. It generalizes the Upper Bound theorem of [15].

(c) Then, we show that $\mathcal{R}^*_E(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\mathcal{B})$, and we get the main completeness theorem: $\mathcal{L}(\mathbb{A}^*_{\mathcal{R},E}) \subseteq \mathcal{R}^*_E(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\mathcal{B})$.

Besides, we know from [15] that $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\mathbb{A}^*_{\mathcal{R},E})$. Thus, when using the set of equations defined from $\mathcal{B}$ to run completion, (c) implies that we can only get an over-approximation of $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ equivalent or better than $\mathscr{L} = \mathcal{L}(\mathcal{B})$. This result has a practical impact for program verification. In particular, for TRSs encoding functional programs, the search space of sets of equations E can be constrained for enumeration to be possible. This has been implemented in the **Timbuk** [12] tool. Our experiments show that this makes completion automatic enough to carry out safety proofs on first-order and higher-order functional programs. We also get a corollary of (c) when $\mathscr{L}$ is *not* an approximation:

(d) If $\mathscr{L} = \mathcal{L}(\mathcal{B}) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$, we can use $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\mathbb{A}^*_{\mathcal{R},E})$ to close-up the $\subseteq$-chain and get that $\mathcal{L}(\mathbb{A}^*_{\mathcal{R},E}) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$. Thus if $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ is regular, there exists a set of equations E s.t. $\mathcal{L}(\mathbb{A}^*_{\mathcal{R},E}) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$

Section 2 defines some basic notions in term rewriting and tree automata and Section 3 recalls the tree automata completion algorithm and the related theorems. Section 4 recalls the Myhill-Nerode theorem for trees and defines the functions to transform a set of equations into a tree automaton and vice versa. Section 5 proves Result (a) and Section 6 shows Result (b). Section 7 assembles (a) and (b) to prove results (c) and (d) using the proof sketched above. Section 8 shows how to take advantage of those results to program verification and presents some experiments. Finally, Section 9 concludes.

2   Preliminaries

In this section we introduce some definitions and concepts that will be used throughout the rest of the paper (see also [2, 6]). Let $\mathcal{F}$ be a finite set of symbols, each associated with an arity function. For brevity, we write $f : n$ if f is a symbol of arity n and $\mathcal{F}^n = \{f \in \mathcal{F} \mid f : n\}$. Let $\mathcal{X}$ be a countable set of *variables*, $\mathcal{T}(\mathcal{F}, \mathcal{X})$ denotes the set of *terms* and $\mathcal{T}(\mathcal{F})$ denotes the set of *ground terms* (terms without variables). The set of variables of a term t is denoted by $\mathscr{V}ar(t)$. A *substitution* is a function σ from $\mathcal{X}$ into $\mathcal{T}(\mathcal{F}, \mathcal{X})$, which can be uniquely extended to an endomorphism of $\mathcal{T}(\mathcal{F}, \mathcal{X})$. A *position* p in a term t is a finite word over $\mathbb{N}$, the set of natural numbers. The empty sequence λ denotes the top-most position. The set $\mathscr{P}os(t)$ of positions of a term t is inductively defined by $\mathscr{P}os(t) = \{\lambda\}$ if $t \in \mathcal{X}$ or t is a constant and $\mathscr{P}os(f(t_1, \ldots, t_n)) = \{\lambda\} \cup \{i.p \mid 1 \leq i \leq n \text{ and } p \in \mathscr{P}os(t_i)\}$ otherwise. If $p \in \mathscr{P}os(t)$, then $t(p)$ denotes the symbol at position p in t, $t|_p$ denotes the subterm of t at

position p, and $t[s]_p$ denotes the term obtained by replacing the subterm $t|_p$ at position p by the term s. A ground context $C[\,]$ is a term in $\mathcal{T}(\mathcal{F} \cup \{\Box\})$ containing exactly one occurrence of the symbol $\Box$. If $t \in \mathcal{T}(\mathcal{F})$ then $C[t]$ denotes the term obtained by the replacement of $\Box$ by t in $C[\,]$. A context is empty if it is equal to $\Box$.

A *term rewriting system* (TRS) $\mathcal{R}$ is a set of *rewrite rules* $l \to r$, where $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $l \notin \mathcal{X}$, and $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$. A rewrite rule $l \to r$ is *left-linear* if each variable occurs only once in l. A TRS $\mathcal{R}$ is left-linear if every rewrite rule $l \to r$ of $\mathcal{R}$ is left-linear. The TRS $\mathcal{R}$ induces a rewriting relation $\to_{\mathcal{R}}$ on terms as follows. Let $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $l \to r \in \mathcal{R}$, $s \to_{\mathcal{R}} t$ denotes that there exists a position $p \in \mathscr{P}os(s)$ and a substitution σ such that $s|_p = l\sigma$ and $t = s[r\sigma]_p$. The set of ground terms irreducible by a TRS $\mathcal{R}$ is denoted by $\mathrm{IRR}(\mathcal{R})$. A set $\mathscr{L} \subseteq \mathcal{T}(\mathcal{F})$ is $\mathcal{R}$-closed if for all $s \in \mathscr{L}$ and $s \to_{\mathcal{R}} t$ then $t \in \mathscr{L}$. The reflexive transitive closure of $\to_{\mathcal{R}}$ is denoted by $\to_{\mathcal{R}}^*$, and $s \to_{\mathcal{R}}^! t$ denotes that $s \to_{\mathcal{R}}^* t$ and t is irreducible by $\mathcal{R}$. The set of $\mathcal{R}$-descendants of a set of ground terms I is defined as $\mathcal{R}^*(I) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in I \text{ s.t. } s \to_{\mathcal{R}}^* t\}$, i.e., the smallest $\mathcal{R}$-closed set containing I. Let E be a *set of equations* $l = r$, where $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. The relation $=_E$ is the smallest congruence such that for all equations $l = r$ of E and for all substitutions σ we have $l\sigma =_E r\sigma$. The set of equivalence classes defined by $=_E$ on $\mathcal{T}(\mathcal{F})$ is denoted by $\mathcal{T}(\mathcal{F})/_{=_E}$. Given a TRS $\mathcal{R}$ and a set of equations E, a term $s \in \mathcal{T}(\mathcal{F})$ is rewritten modulo E into $t \in \mathcal{T}(\mathcal{F})$, denoted $s \to_{\mathcal{R}/E} t$, if there exists an $s' \in \mathcal{T}(\mathcal{F})$ and a $t' \in \mathcal{T}(\mathcal{F})$ such that $s =_E s' \to_{\mathcal{R}} t' =_E t$. The reflexive transitive closure $\to_{\mathcal{R}/E}^*$ of $\to_{\mathcal{R}/E}$ is defined as usual except that reflexivity is extended to terms equal modulo E, i.e., for all $s, t \in \mathcal{T}(\mathcal{F})$, if $s =_E t$ then $s \to_{\mathcal{R}/E}^* t$. The set of $\mathcal{R}$-descendants modulo E of a set of ground terms I is defined as $\mathcal{R}_E^*(I) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in I \text{ s.t. } s \to_{\mathcal{R}/E}^* t\}$.

Let $\mathcal{Q}$ be a countably infinite set of symbols with arity 0, called *states*, such that $\mathcal{Q} \cap \mathcal{F} = \emptyset$. Terms in $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ are called *configurations*. A *transition* is a rewrite rule $c \to q$, where c is a configuration and q is a state. A transition is *normalized* when $c = f(q_1, \ldots, q_n)$, $f \in \mathcal{F}$ is of arity n, and $q_1, \ldots, q_n \in \mathcal{Q}$. An ϵ-*transition* is a transition of the form $q \to q'$ where q and q' are states. A bottom-up non-deterministic finite tree automaton (tree automaton for short) over the alphabet $\mathcal{F}$ is a tuple $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$, where $\mathcal{Q}_f \subseteq \mathcal{Q}$ is the set of final states, Δ is a finite set of normalized transitions and ϵ-transitions. An automaton is *epsilon-free* if it is free of ϵ-transitions. The transitive and reflexive *rewriting relation* on $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ induced by the set of transitions Δ (resp. all transitions except ϵ-transitions) is denoted by $\to_{\Delta}^*$ (resp. $\to_{\Delta}^{\not\epsilon\,*}$). When Δ is attached to a tree automaton $\mathcal{A}$ we also denote those two relations by $\to_{\mathcal{A}}^*$ and $\to_{\mathcal{A}}^{\not\epsilon\,*}$, respectively. A tree automaton $\mathcal{A}$ is complete if for all $s \in \mathcal{T}(\mathcal{F})$ there exists a state q of $\mathcal{A}$ such that $s \to_{\mathcal{A}}^* q$. The *language* recognized by $\mathcal{A}$ in a state q is defined by $\mathcal{L}(\mathcal{A}, q) = \{t \in \mathcal{T}(\mathcal{F}) \mid t \to_{\mathcal{A}}^* q\}$. We define $\mathcal{L}(\mathcal{A}) = \bigcup_{q \in \mathcal{Q}_f} \mathcal{L}(\mathcal{A}, q)$. A state q of an automaton $\mathcal{A}$ is *reachable* if $\mathcal{L}(\mathcal{A}, q) \neq \emptyset$. An automaton is *reduced* if all its states are reachable. An automaton $\mathcal{A}$ is $\not\epsilon$-*reduced* if for all states q of $\mathcal{A}$ there exists a ground term $t \in \mathcal{T}(\mathcal{F})$ such that $t \to_{\mathcal{A}}^{\not\epsilon\,*} q$. An automaton $\mathcal{A}$ is deterministic if for all ground terms $s \in \mathcal{T}(\mathcal{F})$ and all states q, q' of $\mathcal{A}$, if $s \to_{\mathcal{A}}^* q$ and $s \to_{\mathcal{A}}^* q'$ then $q = q'$. An automaton $\mathcal{A}$ is $\mathcal{R}$-closed if for all terms s, t and all states $q \in \mathcal{Q}$, $s \to_{\mathcal{A}}^* q$ and $s \to_{\mathcal{R}} t$ implies $t \to_{\mathcal{A}}^* q$.

3 Equational Tree Automata Completion

From a tree automaton $\mathcal{A}_0 = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta_0 \rangle$ and a left-linear TRS $\mathcal{R}$, the completion algorithm computes an automaton $\mathcal{A}^*$ such that $\mathcal{L}(\mathcal{A}^*) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}_0))$ or $\mathcal{L}(\mathcal{A}^*) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}_0))$.

3.1   Completion General Principles

From $\mathcal{A}_{\mathcal{R}}^0 = \mathcal{A}_0$, tree automata completion successively computes tree automata $\mathcal{A}_{\mathcal{R}}^1$, $\mathcal{A}_{\mathcal{R}}^2$, ...
such that for all $i \geq 0 : \mathcal{L}(\mathcal{A}_{\mathcal{R}}^i) \subseteq \mathcal{L}(\mathcal{A}_{\mathcal{R}}^{i+1})$ and if $s \in \mathcal{L}(\mathcal{A}_{\mathcal{R}}^i)$, and $s \rightarrow_{\mathcal{R}} t$ then $t \in \mathcal{L}(\mathcal{A}_{\mathcal{R}}^{i+1})$.
For $k \in \mathbb{N}$, if $\mathcal{L}(\mathcal{A}_{\mathcal{R}}^k) = \mathcal{L}(\mathcal{A}_{\mathcal{R}}^{k+1})$ then $\mathcal{A}_{\mathcal{R}}^k$ is a fixpoint and we denote it by $\mathcal{A}_{\mathcal{R}}^*$. To construct
$\mathcal{A}_{\mathcal{R}}^{i+1}$ from $\mathcal{A}_{\mathcal{R}}^i$, we perform a *completion step* (denoted by $\mathcal{C}_{\mathcal{R}}$) which consists in finding
critical pairs between $\rightarrow_{\mathcal{R}}$ and $\rightarrow_{\mathcal{A}_{\mathcal{R}}^i}$. For a substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ and a rule $l \rightarrow r \in \mathcal{R}$, a
critical pair is an instance $l\sigma$ of l such that there exists a state $q \in \mathcal{Q}$ satisfying $l\sigma \rightarrow_{\mathcal{A}_{\mathcal{R}}^i}^* q$
and $r\sigma \not\rightarrow_{\mathcal{A}_{\mathcal{R}}^i}^* q$. For $r\sigma$ to be recognized by the same state and thus model the rewriting
of $l\sigma$ into $r\sigma$, it is enough to add the necessary transitions to $\mathcal{A}_{\mathcal{R}}^i$ in order to obtain $\mathcal{A}_{\mathcal{R}}^{i+1}$
such that $r\sigma \rightarrow_{\mathcal{A}_{\mathcal{R}}^{i+1}}^* q$. The result of the completion step $\mathcal{C}_{\mathcal{R}}(\mathcal{A}_{\mathcal{R}}^i)$ is thus $\mathcal{A}_{\mathcal{R}}^{i+1}$. In [24, 15],
critical pairs are joined as in Figure 1.
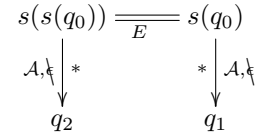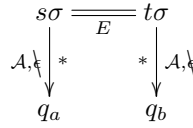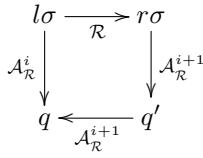
Figure 1 Completion step   **Figure 2** Simplification step   **Figure 3** Simplification example

From an algorithmic point of view, there remain two problems to solve: find all the critical
pairs $(l \rightarrow r, \sigma, q)$ and find the transitions to add to $\mathcal{A}_{\mathcal{R}}^i$ to have $r\sigma \rightarrow_{\mathcal{A}_{\mathcal{R}}^{i+1}}^* q$. The first
problem, called matching, can be efficiently solved using a specific algorithm [8]. The second
problem is solved using a normalization algorithm [10]. To have $r\sigma \rightarrow_{\mathcal{A}_{\mathcal{R}}^{i+1}}^* q'$ we need a
transition of the form $r\sigma \rightarrow q'$ in $\mathcal{A}_{\mathcal{R}}^{i+1}$. However, this transition may not be normalized.
In this case, it is necessary to introduce new states and new transitions. For instance,
to normalize a transition $f(g(a), h(q_1)) \rightarrow q'$ w.r.t. a tree automaton $\mathcal{A}_{\mathcal{R}}^i$ with transitions
$a \rightarrow q_1$, $b \rightarrow q_1$, $g(q_1) \rightarrow q_1$, we first rewrite $f(g(a), h(q_1))$ with transitions of $\mathcal{A}_{\mathcal{R}}^i$ as far
as possible. We obtain $f(q_1, h(q_1))$. Then we introduce the new state q_2 and the new
transition $h(q_1) \rightarrow q_2$ to recognize the term $h(q_1)$. The new transitions to add to $\mathcal{A}_{\mathcal{R}}^i$ are
thus: $h(q_1) \rightarrow q_2$, $f(q_1, q_2) \rightarrow q'$, and $q' \rightarrow q$.

3.2   Simplification of Tree Automata by Equations

Since completion creates new transitions and new states to join critical pairs, it may diverge.
Divergence is avoided by *simplifying* the tree automaton with a set of equations E. This
operation permits the over-approximation of languages that cannot be recognized *exactly*
using tree automata completion, e.g., non-regular languages. Simplification consists in finding
E-equivalent terms recognized in $\mathcal{A}$ by different states and then by merging those states.

▶ **Definition 1** (Simplification relation). Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton and E
be a set of equations. For $s = t \in E$, $\sigma : \mathcal{X} \mapsto \mathcal{Q}$, $q_a, q_b \in \mathcal{Q}$ such that $s\sigma \rightarrow_{\mathcal{A}}^* q_a$, $t\sigma \rightarrow_{\mathcal{A}}^* q_b$
(See Figure 2) and $q_a \neq q_b$ then $\mathcal{A}$ is *simplified* into $\mathcal{A}'$, denoted by $\mathcal{A} \leadsto_E \mathcal{A}'$, where $\mathcal{A}'$ is
$\mathcal{A}$ where q_b is replaced by q_a in $\mathcal{Q}$, $\mathcal{Q}_f$ and Δ.   ◇

▶ **Example 2.** Let $E = \{s(s(x)) = s(x)\}$ and $\mathcal{A}$ be the tree automaton with $\mathcal{Q}_f = \{q_2\}$
and set of transitions $\Delta = \{a \rightarrow q_0, s(q_0) \rightarrow q_1, s(q_1) \rightarrow q_2\}$. Hence $\mathcal{L}(\mathcal{A}) = \{s(s(a))\}$. We
can perform a simplification step using the equation $s(s(x)) = s(x)$ because we found a
substitution $\sigma = \{x \mapsto q_0\}$ such that $s(s(q_0)) \rightarrow_{\mathcal{A}}^* q_2$, $s(q_0) \rightarrow_{\mathcal{A}}^* q_1$ (see Figure 3). Hence,

$\mathcal{A} \leadsto_E \mathcal{A}'$ where $\mathcal{A}'$ is $\mathcal{A}$ where q_2 is replaced by q_1 i.e., $\mathcal{A}'$ is the automaton with $\mathcal{Q}'_f = \{q_1\}$, $\Delta = \{a \to q_0, s(q_0) \to q_1, s(q_1) \to q_1\}$. Note that $\mathcal{L}(\mathcal{A}') = \{s^*(s(a))\}$.

The simplification relation $\leadsto_E$ is terminating and confluent (modulo state renaming) [15]. In the following, by $\mathcal{S}_E(\mathcal{A})$ we denote the unique automaton (modulo renaming) $\mathcal{A}'$ such that $\mathcal{A} \leadsto_E^* \mathcal{A}'$ and $\mathcal{A}'$ is irreducible (it cannot be simplified further).

3.3 The full Completion Algorithm

▶ **Definition 3** (Automaton completion). Let $\mathcal{A}$ be a tree automaton, $\mathcal{R}$ a left-linear TRS and E a set of equations.

- $\mathcal{A}^0_{\mathcal{R},E} = \mathcal{A}$,
- $\mathcal{A}^{n+1}_{\mathcal{R},E} = \mathcal{S}_E(\mathcal{C}_{\mathcal{R}}(\mathcal{A}^n_{\mathcal{R},E}))$, for $n \geq 0$ where $\mathcal{C}_{\mathcal{R}}(\mathcal{A}^n_{\mathcal{R},E})$ is the tree automaton such that all critical pairs of $\mathcal{A}^n_{\mathcal{R},E}$ are joined.

If there exists $k \in \mathbb{N}$ such that $\mathcal{A}^k_{\mathcal{R},E} = \mathcal{A}^{k+1}_{\mathcal{R},E}$, then we write $\mathcal{A}^*_{\mathcal{R},E}$ for $\mathcal{A}^k_{\mathcal{R},E}$.

▶ **Example 4.** Let $\mathcal{R} = \{f(x,y) \to f(s(x), s(y))\}$, $E = \{s(s(x)) = s(x)\}$ and $\mathcal{A}^0$ be the tree automaton with set of transitions $\Delta = \{f(q_a, q_b) \to q_0, a \to q_a, b \to q_b\}$, i.e., $\mathcal{L}(\mathcal{A}^0) = \{f(a,b)\}$. The completion ends after two completion steps on $\mathcal{A}^2_{\mathcal{R},E}$ which is a fixpoint $\mathcal{A}^*_{\mathcal{R},E}$. Completion steps are summed up in the following table. To simplify the presentation, we do not repeat the common transitions: $\mathcal{A}^i_{\mathcal{R},E}$ and $\mathcal{C}_{\mathcal{R}}(\mathcal{A}^i)$ columns are supposed to contain all transitions of $\mathcal{A}^0, \ldots, \mathcal{A}^{i-1}_{\mathcal{R},E}$.

$\mathcal{A}^0$	$\mathcal{C}_{\mathcal{R}}(\mathcal{A}^0)$	$\mathcal{A}^1_{\mathcal{R},E}$	$\mathcal{C}_{\mathcal{R}}(\mathcal{A}^1_{\mathcal{R},E})$	$\mathcal{A}^2_{\mathcal{R},E}$
$f(q_a, q_b) \to q_0$	$f(q_1, q_2) \to q_3$	$f(q_1, q_2) \to q_3$	$f(q_4, q_5) \to q_6$	$f(q_1, q_2) \to q_6$
$a \to q_a$	$s(q_a) \to q_1$	$s(q_a) \to q_1$	$s(q_1) \to q_4$	$s(q_1) \to q_1$
$b \to q_b$	$s(q_b) \to q_2$	$s(q_b) \to q_2$	$s(q_2) \to q_5$	$s(q_2) \to q_2$
	$q_3 \to q_0$	$q_3 \to q_0$	$q_6 \to q_3$	

On $\mathcal{A}^0$, there is one critical pair $f(q_a, q_b) \to_{\mathcal{A}_0}^* q_0$ and $f(q_a, q_b) \to_{\mathcal{R}} f(s(q_a), s(q_b))$. The automaton $\mathcal{C}_{\mathcal{R}}(\mathcal{A}^0)$ contains all the transitions of $\mathcal{A}^0$ with the new transitions (and the new states) necessary to join the critical pair, i.e., to have $f(s(q_a), s(q_b)) \to_{\mathcal{C}_{\mathcal{R}}(\mathcal{A}^0)}^* q_0$. The automaton $\mathcal{A}^1_{\mathcal{R},E}$ is exactly $\mathcal{C}_{\mathcal{R}}(\mathcal{A}^0)$ because simplification by E does not apply. Then, $\mathcal{C}_{\mathcal{R}}(\mathcal{A}^1_{\mathcal{R},E})$ contains all the transitions of $\mathcal{A}^1_{\mathcal{R},E}$ and $\mathcal{A}^0$ plus those obtained by the resolution of the critical pair $f(q_1, q_2) \to_{\mathcal{A}^1_{\mathcal{R},E}}^* q_3$ and $f(q_1, q_2) \to_{\mathcal{R}} f(s(q_1), s(q_2))$. On $\mathcal{C}_{\mathcal{R}}(\mathcal{A}^1_{\mathcal{R},E})$ simplification using the equation $s(s(x)) = s(x)$ can be applied on the following instances: $s(s(q_a)) = s(q_a)$ and $s(s(q_b)) = q_b$. Since $s(s(q_a)) \to_{\mathcal{C}_{\mathcal{R}}(\mathcal{A}^1_{\mathcal{R},E})}^* q_4$ and $s(q_a) \to_{\mathcal{C}_{\mathcal{R}}(\mathcal{A}^1_{\mathcal{R},E})}^* q_1$, simplification merges q_4 with q_1. Similarly, simplification on $s(s(q_b)) = q_b$ merges q_5 with q_2. Thus, $\mathcal{A}^2_{\mathcal{R},E} = \mathcal{C}_{\mathcal{R}}(\mathcal{A}^1_{\mathcal{R},E})$ where q_4 is replaced by q_1 and q_5 is replaced by q_2. This automaton is a fixed point because it has no other critical pairs (they are all joined).

3.4 Lower Bound, Upper Bound and Termination of Completion

▶ **Theorem 5** (Lower Bound [15]). *Let $\mathcal{R}$ be a left-linear TRS, $\mathcal{A}$ be a tree automaton and E be a set of equations. If completion terminates on $\mathcal{A}^*_{\mathcal{R},E}$ then $\mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$.*

To state the upper bound theorem, we need the notion of $\mathcal{R}/E$-coherence we now define.

▶ **Definition 6** (Coherent automaton). Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton, $\mathcal{R}$ a TRS and E a set of equations. The automaton $\mathcal{A}$ is said to be $\mathcal{R}/E$-coherent if $\forall q \in \mathcal{Q}$: $\exists s \in \mathcal{T}(\mathcal{F})$:

$$s \rightarrow_{\mathcal{A}}^{\not{\epsilon}*} q \wedge [\forall t \in \mathcal{T}(\mathcal{F}) : (t \rightarrow_{\mathcal{A}}^{\not{\epsilon}*} q \implies s =_E t) \wedge (t \rightarrow_{\mathcal{A}}^{*} q \implies s \rightarrow_{\mathcal{R}/E}^{*} t)].$$

Here is the intuition behind $\mathcal{R}/E$-coherence. An $\mathcal{R}/E$-coherent automaton is $\not{\epsilon}$-reduced, its ϵ-transitions represent rewriting steps and normalized ($\not{\epsilon}$-transitions) transitions recognize E-equivalence classes. More precisely, in an $\mathcal{R}/E$-coherent tree automaton, if two terms s, t are recognized in the same state q using only normalized transitions then they belong to the same E-equivalence class. Otherwise, if at least one ϵ-transition is necessary to recognize, say, t in q then at least one step of rewriting with $\mathcal{R}$ was necessary to obtain t from s.

▶ **Example 7.** Let $\mathcal{R} = \{a \rightarrow b\}$, $E = \{c = d\}$ and $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ with $\Delta = \{a \rightarrow q_0, b \rightarrow q_1, c \rightarrow q_2, d \rightarrow q_2, q_1 \rightarrow q_0\}$. The automaton $\mathcal{A}$ is $\mathcal{R}/E$-coherent because it is $\not{\epsilon}$-reduced and the state q_2 recognizes with $\rightarrow_{\Delta}^{\not{\epsilon}}$ two terms c and d but they satisfy $c =_E d$. Finally, $a \rightarrow_{\Delta}^{*} q_0$ and $b \rightarrow_{\Delta}^{*} q_0$ but $a \rightarrow_{\Delta}^{\not{\epsilon}} q_0$, $b \rightarrow_{\Delta}^{\not{\epsilon}} q_1 \rightarrow q_0$ and $a \rightarrow_{\mathcal{R}} b$.

▶ **Theorem 8** (Upper Bound [15]). *Let $\mathcal{R}$ be a left-linear TRS, E a set of equations and $\mathcal{A}$ an $\mathcal{R}/E$-coherent automaton. For any $i \in \mathbb{N}$: $\mathcal{L}(\mathcal{A}_{\mathcal{R},E}^i) \subseteq \mathcal{R}_E^*(\mathcal{L}(\mathcal{A}))$ and $\mathcal{A}_{\mathcal{R},E}^i$ is $\mathcal{R}/E$-coherent.*

Finally, we state the termination theorem which relies on E-compatibility. Roughly speaking, E-compatibility is the symmetric of E-coherence. An automaton $\mathcal{A}$ is E-compatible if for all states $q_1, q_2 \in \mathcal{A}$ and all terms $s, t \in \mathcal{T}(\mathcal{F})$ such that $s \rightarrow_{\mathcal{A}}^{\not{\epsilon}*} q_1$, $t \rightarrow_{\mathcal{A}}^{\not{\epsilon}*} q_2$ and $s =_E t$ then we have $q_1 = q_2$.

▶ **Theorem 9** (Termination of completion [10]). *Let $\mathcal{A}$ be a $\not{\epsilon}$-reduced tree automaton, $\mathcal{R}$ a left-linear TRS, and E a set of equations such that $\mathcal{T}(\mathcal{F})/_{=_E}$ is finite. If for all $i \in \mathbb{N}$, $\mathcal{A}_{\mathcal{R},E}^i$ is E-compatible then there exists a natural number $k \in \mathbb{N}$ such that $\mathcal{A}_{\mathcal{R},E}^k$ is a fixpoint.*

To prove our final result, we first have to generalize Theorems 8 and 9 to discard the technical $\mathcal{R}/E$-coherence and E-compatibility assumptions. This is the objective of the next sections.

4   From automata to equations and vice versa

Theorem 9 uses the assumption that the automata $\mathcal{A}_{\mathcal{R},E}^i$ are all E-compatible. This is not true in general. Unlike $\mathcal{R}/E$-coherence, E-compatibility is not preserved by tree automaton completion: $\mathcal{A}_{\mathcal{R},E}^{i+1}$ may not be E-compatible even if $\mathcal{A}_{\mathcal{R},E}^i$ is. Proofs can be found in [11].

▶ **Example 10.** Let $\mathcal{F} = \{f : 1, a : 0, b : 0, c : 0\}$, $\mathcal{R} = \{f(x) \rightarrow f(f(x)), f(f(x)) \rightarrow a\}$, $\mathcal{A}$ be the automaton such that $\Delta = \{a \rightarrow q_1, c \rightarrow q_1, f(q_1) \rightarrow q_f\}$ and $E = \{f(a) = f(b), f(b) = b, f(c) = f(b)\}$. Note that $\mathcal{T}(\mathcal{F})/_{=_E}$ has 3 equivalence classes: the class of $\{a\}$, the class of $\{b, f(a), f(b), f(c), \ldots\}$ and the class of $\{c\}$. However, completion does not terminate on this example. Automaton $\mathcal{A}$ is E-compatible ($f(a) =_E f(c)$ and both terms are recognized with $\rightarrow_{\mathcal{A}}^{\not{\epsilon}}$ by the same state: q_f) but $\mathcal{A}_{\mathcal{R},E}^1$ is not: it has one new state q_2 and contains additional transitions $\{f(q_f) \rightarrow q_2, q_2 \rightarrow q_f\}$. We thus have $f(f(a)) \rightarrow_{\mathcal{A}_{\mathcal{R},E}^1}^{\not{\epsilon}*} q_2$ and $f(a) \rightarrow_{\mathcal{A}_{\mathcal{R},E}^1}^{\not{\epsilon}*} q_f$ and $f(f(a)) =_E f(f(b)) =_E f(b) =_E f(a)$ but $q_2 \neq q_f$. Since b is not recognized by $\mathcal{A}_{\mathcal{R},E}^n$ for any n, the equation $f(b) = b$ never applies and completion diverges.

E-compatibility can be ensured for particular cases of $\mathcal{R}$ and E, e.g., for typed functional programs [10]. Here, we show how to transform the set E into a set $E_{\mathcal{B}}$ for which completed automata are $E_{\mathcal{B}}$-compatible, and completion is thus terminating. We also build $E_{\mathcal{B}}$ so that its precision is similar to E, i.e., $=_E \equiv =_{E_{\mathcal{B}}}$. This transformation is based on the Myhill-Nerode theorem for trees [18, 6]. We first produce a tree automaton $\mathcal{B}$ whose states recognize the equivalence classes of E. Then, from $\mathcal{B}$, we perform the inverse operation and obtain a set $E_{\mathcal{B}}$ whose set of equivalence classes is similar to the classes of E, but whose equations avoid the problem shown in Example 10. In this paper, we mainly consider sets E of ground equations because they are sufficient to prove our completeness results and for the practical applications of Section 7. However, this can be extended to general equations if E can be oriented into a weakly terminating TRS $\mathcal{R}$ s.t. $\text{IRR}(\mathcal{R})$ is finite [11].

4.1 From equations to automata

If $\mathcal{T}(\mathcal{F})/_{=_E}$ is finite, the Myhill-Nerode theorem for trees [18, 6] relates $\mathcal{T}(\mathcal{F})/_{=_E}$ with tree automata. This theorem is constructive and provides an algorithm to switch from one form to the other, provided that $=_E$ is decidable. In the following we denote by **MN** the function that builds a tree automaton from a set of equations E [18].

▶ **Definition 11** (Function **MN**). Let E be a set of equations such that $\mathcal{T}(\mathcal{F})/_{=_E}$ is finite and $=_E$ is decidable. Let $\mathcal{Q}$ be a set of states and $state : \mathcal{T}(\mathcal{F})/_{=_E} \mapsto \mathcal{Q}$ be an injective function. $\mathbf{MN}(E) = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}, \Delta \rangle$ where $\Delta = \{ f(state(u_1), \dots, state(u_n)) \to state(u) \mid f \in \mathcal{F}, u_1, \dots, u_n, u \in \mathcal{T}(\mathcal{F})/_{=_E} \text{ and } f(u_1, \dots, u_n) =_E u \}$

▶ **Theorem 12** (Myhill-Nerode theorem for trees [18]). *If $\mathcal{T}(\mathcal{F})/_{=_E}$ is finite and $=_E$ decidable, $\mathcal{B} = \mathbf{MN}(E)$ is a reduced, deterministic, epsilon-free and complete tree automaton such that for all $s, t \in \mathcal{T}(\mathcal{F})$, $s =_E t \iff (\exists q : \{s, t\} \subseteq \mathcal{L}(\mathcal{B}, q))$.*

When all equations of E are ground, E can be oriented into a complete TRS (confluent and terminating) $\overrightarrow{E}$, using for instance [22]. Then $=_E$ is decidable using $\overrightarrow{E}$ and finiteness of $\mathcal{T}(\mathcal{F})/_{=_E}$ is equivalent to finiteness of $\text{IRR}(\overrightarrow{E})$, which is decidable [6].

▶ **Example 13.** Consider the set E of Example 10. We can orient E into a complete TRS $\overrightarrow{E} = \{ f(a) \to f(b), f(b) \to b, f(c) \to f(b) \}$. The set $\text{IRR}(\overrightarrow{E})$ is $\{a, b, c\}$. The automaton $\mathbf{MN}(E)$ has 3 states q_0, q_1, q_2 such that $state(a) = q_0$, $state(b) = q_1$ and $state(c) = q_2$. It has six transitions $a \to q_0$ (because $a \to^!_{\overrightarrow{E}} a$), $b \to q_1$ (because $b \to^!_{\overrightarrow{E}} b$), $c \to q_2$ (because $c \to^!_{\overrightarrow{E}} c$), $f(q_0) \to q_1$ (because $f(a) \to^!_{\overrightarrow{E}} b$), $f(q_1) \to q_1$ (because $f(b) \to^!_{\overrightarrow{E}} b$), $f(q_2) \to q_1$ (because $f(c) \to^!_{\overrightarrow{E}} b$).

4.2 From automata to equations

In the other direction, starting from a tree automaton $\mathcal{B}$ it is possible to build a set of equations $E_{\mathcal{B}}$ such that languages recognized by states of $\mathcal{B}$ and equivalence classes of $\mathcal{T}(\mathcal{F})/_{=_{E_{\mathcal{B}}}}$ coincide [18]. We reformulate the original algorithm into a function called **A2E** because we need some additional properties on the generated set of equations for completion to terminate. For simplicity we assume that $\mathcal{B}$ is **R**educed and epsilon-**F**ree. Some properties of $E_{\mathcal{B}}$ will hold only if $\mathcal{B}$ is also **C**omplete and **D**eterministic. In the following, we use the **RF** and **RDFC** short-hands for automata having the related properties. Recall that for any tree automaton, there exists an equivalent **RF** or **RDFC** automaton [6].

For an **RF** automaton $\mathcal{B}$, the construction of $E_{\mathcal{B}} = \mathbf{A2E}(\mathcal{B})$ is straightforward and follows [18]: for all states q we identify a ground term recognized by q, a representative,

and for all transitions $f(q_1, \ldots, q_n) \to q$ we generate an equation $f(t_1, \ldots, t_n) = t$ where t_i, $1 \leq i \leq n$ are representatives for q_i and t is a representative for q. However, for this set of equations to guarantee termination of completion it needs some redundancy: for each state we generate a *set* of *state representatives* and the equations are defined for each representative of the set. As shown in Example 10, the equation $f(b) = b$ cannot be applied during completion because b does not occur in the tree automaton. However, a logical consequence of this equation is that $f(f(a)) =_E f(a)$ and terms $f(f(a))$ and $f(a)$ that occur in the tree automaton could be merged. In our setting the term $f(a)$ will be a state representative and the equation $f(f(a)) = f(a)$ will appear in the set of generated equations. Roughly speaking, every constant symbol a appearing in a transition $a \to q$ is a state representative for q. Every term of the form $f(u_1, \ldots, u_n)$ is a state representative for q if (1) u_i's are not state representatives of q, (2) $f(q_1, \ldots, q_n) \to q$ is a transition of $\mathcal{B}$ and (3) u_i's are state representatives for the q_i's. The property (1) ensures finiteness of the set of representatives.

▶ **Definition 14** (State representatives). Let $\mathcal{B} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be an **RF** tree automaton and $q \in \mathcal{Q}$. The set of state representatives of q of height lesser or equal to $k \in \mathbb{N}$, denoted by $[\![q]\!]_{\mathcal{B}}^k$, is inductively defined by:

- $[\![q]\!]_{\mathcal{B}}^1 = \{a \mid a \to q \in \Delta\}$
- $[\![q]\!]_{\mathcal{B}}^k = [\![q]\!]_{\mathcal{B}}^{k-1} \cup \{f(u_1, \ldots, u_n) \mid f(q_1, \ldots, q_n) \to q \in \Delta \text{ and } 1 \leq i \leq n, u_i \in [\![q_i]\!]_{\mathcal{B}}^{k-1},$
  $\text{and } \forall p \in \mathscr{P}os(u_i) : u_i|_p \notin [\![q]\!]_{\mathcal{B}}^{k-1}\}$

In the above definition, the fact that $\mathcal{B}$ is reduced and epsilon-free ensures that there exists at least one (non-epsilon) transition for every state and that each state has at least one state representative.

▶ **Example 15.** Let $\mathcal{B}$ be the **RF** automaton that we obtained in Example 13 and whose set of transitions is $a \to q_0$, $b \to q_1$, $c \to q_2$, $f(q_0) \to q_1$, $f(q_1) \to q_1$, $f(q_2) \to q_1$.

- $[\![q_0]\!]_{\mathcal{B}}^1 = \{a\}$, $[\![q_1]\!]_{\mathcal{B}}^1 = \{b\}$, and $[\![q_2]\!]_{\mathcal{B}}^1 = \{c\}$.
- $[\![q_0]\!]_{\mathcal{B}}^2 = [\![q_0]\!]_{\mathcal{B}}^1$, $[\![q_1]\!]_{\mathcal{B}}^2 = \{b, f(a), f(c)\}$, and $[\![q_2]\!]_{\mathcal{B}}^2 = [\![q_2]\!]_{\mathcal{B}}^1$. The term $f(b)$ of height 2 and recognized by q_1 is not added to $[\![q_1]\!]_{\mathcal{B}}^2$ because its subterm b belongs to $[\![q_1]\!]_{\mathcal{B}}^1$.
- The fixpoint is reached because terms $f(f(a))$ and $f(f(c))$ recognized by q_1 are not added to $[\![q_1]\!]_{\mathcal{B}}^3$ because $f(a)$ and $f(c)$ belong to $[\![q_1]\!]_{\mathcal{B}}^2$.

We denote by $[\![q]\!]_{\mathcal{B}}$ the set of all state representatives for the state q i.e., the fixpoint of the above equations. We know that such a fixpoint exists and is always a finite set. Omitted proofs can be found in [11].

▶ **Lemma 16** (The set of state representatives is finite). *For all **RF** tree automata $\mathcal{B}$, for all states $q \in \mathcal{B}$ there exists a natural number $k \in \mathbb{N}$ for which the set $[\![q]\!]_{\mathcal{B}}^k$ is a fixpoint.*

▶ **Definition 17** (Function **A2E**: set of equations $E_{\mathcal{B}}$ from a tree automaton $\mathcal{B}$). Let $\mathcal{B} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be an **RF** tree automaton. The set of equations $E_{\mathcal{B}}$ inferred from $\mathcal{B}$ is $\mathbf{A2E}(\mathcal{B}) = \{f(u_1, \ldots, u_n) = u \mid f(q_1, \ldots, q_n) \to q \in \mathcal{B}, u \in [\![q]\!]_{\mathcal{B}} \text{ and } u_i \in [\![q_i]\!]_{\mathcal{B}} \text{ for } 1 \leq i \leq n\}$.

▶ **Example 18.** Starting from the automaton $\mathcal{B}$ and the state representatives of Example 15, the set $\mathbf{A2E}(\mathcal{B})$ contains the following equations: $a = a$ (because of transition $a \to q_0$), $c = c$ (because of transition $c \to q_2$), $b = b$, $b = f(a)$, $b = f(c)$ (because of transition $b \to q_1$), $f(a) = f(a)$, $f(a) = b$, $f(a) = f(c)$ (because of transition $f(q_0) \to q_1$), $f(f(a)) = f(a)$, $f(f(a)) = b$, $f(f(a)) = f(c)$, $f(b) = f(a)$, $f(b) = b$, $f(b) = f(c)$, $f(f(c)) = f(a)$, $f(f(c)) = b$, $f(f(c)) = f(c)$ (because of transition $f(q_1) \to q_1$), $f(c) = f(a)$, $f(c) = b$, and $f(c) = f(c)$ (because of transition $f(q_2) \to q_1$).

Since $\mathcal{B}$ is finite and the set of state representatives is finite then so is $E_{\mathcal{B}}$. Note that many equations of $E_{\mathcal{B}}$ are useless w.r.t. the underlying equational theory. This is the case, in the above example, for equations of the form $a = a$ as well as the equation $f(a) = f(c)$ which is redundant w.r.t. $b = f(a)$ and $b = f(c)$. However, as shown in Example 10 those equations are necessary for equational simplification to produce $E_{\mathcal{B}}$-compatible automata and completion to terminate. With the above $E_{\mathcal{B}}$, completion of Example 10 terminates. Below, Theorem 23 shows that, if $\mathcal{B}$ is **RDFC** then completion with **A2E**($\mathcal{B}$) always terminates. Unsurprisingly, if $\mathcal{B}$ is deterministic then equivalence classes of $E_{\mathcal{B}}$ coincide with languages recognized by states of $\mathcal{B}$. This is the purpose of the next two lemmas.

▶ **Lemma 19.** *Let $\mathcal{B} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be an* **RDFC** *tree automaton and $E_{\mathcal{B}} = $* **A2E**($\mathcal{B}$). *For all $s \in \mathcal{T}(\mathcal{F})$, there exists a unique state $q \in \mathcal{Q}$ such that $s \to_{\mathcal{B}}^* q$ and for all state representatives $u \in [\![q]\!]_{\mathcal{B}}$, $s =_{E_{\mathcal{B}}} u$.*

Now we can relate equivalence classes of $E_{\mathcal{B}}$ and languages recognized by states of $\mathcal{B}$.

▶ **Lemma 20** (Equivalence classes of $E_{\mathcal{B}}$ coincide with languages recognized by states of $\mathcal{B}$). *Let $\mathcal{B} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be an* **RDFC** *tree automaton and $E_{\mathcal{B}} = $* **A2E**($\mathcal{B}$). *For all $s, t \in \mathcal{T}(\mathcal{F})$, $s =_{E_{\mathcal{B}}} t \iff (\exists q : \{s, t\} \subseteq \mathcal{L}(\mathcal{B}, q))$.*

▶ **Corollary 21** ($\mathcal{T}(\mathcal{F})/_{=_{E_{\mathcal{B}}}}$ is finite). *Let $\mathcal{B} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be an* **RDFC** *tree automaton. If $E_{\mathcal{B}}$ is the set of equations inferred from $\mathcal{B}$ then $\mathcal{T}(\mathcal{F})/_{=_{E_{\mathcal{B}}}}$ is finite.*

5 Generalizing the termination theorem

Now, we prove that using $E_{\mathcal{B}}$ built from an **RDFC** tree automaton $\mathcal{B}$, completion terminates.

5.1 Proving termination of completion with $E_{\mathcal{B}}$

In the following, the automaton $\mathcal{A}^*$ is the limit of the (possibly) infinite completion of an initial ε-reduced tree automaton $\mathcal{A}$ with $\mathcal{R}$ and $E_{\mathcal{B}}$. If the initial automaton is not ε-reduced then completion may diverge. For instance, completion of the automaton whose set of transitions is $\{f(q_0) \to q_1\}$, with $\mathcal{R} = \{f(x) \to f(f(x))\}$ and $E = \{f(a) = a\}$ diverges (simplification never happens because q_0 does not recognize any term). Now we show that all state representatives are recognized by epsilon-free derivations in $\mathcal{A}^*$.

▶ **Lemma 22** (All states of $\mathcal{A}^*$ recognize at least one state representative). *Let $\mathcal{R}$ be a TRS, $\mathcal{A}$ a ε-reduced tree automaton, $\mathcal{B}$ an* **RDFC** *tree automaton and $E_{\mathcal{B}} = $* **A2E**($\mathcal{B}$). *Let $\mathcal{A}^*$ be the limit of the completion of $\mathcal{A}$ by $\mathcal{R}$ and $E_{\mathcal{B}}$. For all states $q \in \mathcal{A}^*$, for all terms $s \in \mathcal{T}(\mathcal{F})$ such that $s \to_{\mathcal{A}^*}^{\varepsilon *} q$, there exists a state $q_{\mathcal{B}}' \in \mathcal{B}$, a term $u \in [\![q_{\mathcal{B}}']\!]_{\mathcal{B}}$ such that $u =_{E_{\mathcal{B}}} s$ and $u \to_{\mathcal{A}^*}^{\varepsilon *} q$.*

Now, we can state the termination theorem with $E_{\mathcal{B}}$.

▶ **Theorem 23** (Completion with $E_{\mathcal{B}}$ terminates). *Let $\mathcal{R}$ be a TRS, $\mathcal{A}$ a ε-reduced tree automaton, $\mathcal{B}$ be an* **RDFC** *tree automaton and $E_{\mathcal{B}} = $* **A2E**($\mathcal{B}$). *Let n be the number of all states representatives of $\mathcal{B}$. The automaton $\mathcal{A}^*$, limit of the completion of $\mathcal{A}$ with $\mathcal{R}$ and $E_{\mathcal{B}}$, has n states or less.*

5.2   Building $E_\mathcal{B}$ from any set of equations E

Now, we combine the transformations **A2E** and **MN** to produce a set of equations $E_\mathcal{B}$ that ensures termination of completion. Unsurprisingly, $E_\mathcal{B}$ is equivalent to E.

▶ **Lemma 24.** *Let E be a set of equations. If $\mathcal{T}(\mathcal{F})/_{=_E}$ is finite and $=_E$ is decidable then $E_\mathcal{B} = \boldsymbol{A2E}(\boldsymbol{MN}(E))$ and $=_E \equiv =_{E_\mathcal{B}}$.*

▶ **Theorem 25** (Generalized termination theorem for completion). *Let E be a set of ground equations such that $\mathcal{T}(\mathcal{F})/_{=_E}$ is finite. For all ϵ-reduced tree automata $\mathcal{A}$ and TRSs $\mathcal{R}$, completion of $\mathcal{A}$ with $\mathcal{R}$ and $\boldsymbol{A2E}(\boldsymbol{MN}(E))$ terminates.*

Proof. As mentioned in Section 4.1, since E is ground $=_E$ is decidable. By Theorem 12, we know that $\mathcal{B} = \boldsymbol{MN}(E)$ exists and is **RDFC**. Let $E_\mathcal{B}$ be the set of equations **A2E**($\mathcal{B}$). Using Theorem 23, we know that completion of $\mathcal{A}$ with $\mathcal{R}$ and $E_\mathcal{B}$ is terminating.   ◀

The above theorem shows how to tune a set of equations E into $E_\mathcal{B}$ to guarantee termination of completion. Note that tuning E into $E_\mathcal{B}$ does not jeopardize the precision of the completion since Lemma 24 guarantees that $=_E \equiv =_{E_\mathcal{B}}$. Combining this lemma with Theorem 8 (the Upper Bound Theorem) yields that completion of $\mathcal{R}$ with $E_\mathcal{B}$ is upper-bounded by $\mathcal{R}_E^*$.

6   Improving the Precision of Equational completion

Looking at our overall goal, we are half way there. If $\mathscr{L}$ is regular and $\mathscr{L} \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ (or $\mathscr{L} = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$) then it can be recognized by an automaton $\mathcal{B}$. Using the results of the last section, we can build a set of equations $E_\mathcal{B}$ guaranteeing termination of completion. What remains to be proved is that completion with $E_\mathcal{B}$ ends on a tree automaton under-approximating $\mathscr{L}$ (or recognizing exactly $\mathscr{L} = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$). As it is, Theorem 8 (the Upper Bound Theorem) fails to tackle this goal because it needs $\mathcal{R}/E$-coherence of $\mathcal{A}$. However, if $\mathcal{A}$ is not $\mathcal{R}/E$-coherent the full precision, granted by this theorem, may not be obtained.

▶ **Example 26.** Starting from Example 10, together with the set of equations $E_\mathcal{B}$ of Example 18, the initial tree automaton is not $\mathcal{R}/E_\mathcal{B}$-coherent (nor $\mathcal{R}/E$-coherent): $a \to_{\mathcal{A}}^{\epsilon *} q_1$ and $c \to_{\mathcal{A}}^{\epsilon *} q_1$ though $a \neq_{E_\mathcal{B}} c$. As a consequence, if we complete $\mathcal{A}$ with $\mathcal{R}$ and $E_\mathcal{B}$, we obtain an automaton that roughly approximates $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$. This can be done using the Timbuk tool [12]:

States q0 q1 **Final States** q0 **Transitions** c->q1 a->q1 c->q0 f(q0)->q0 f(q1)->q0 a->q0

This automaton recognizes the term c that is not reachable by rewriting the initial language $\mathcal{L}(\mathcal{A}) = \{f(a), f(c)\}$ with $\mathcal{R}$ (nor by rewriting with $\mathcal{R}/E_\mathcal{B}$). We propose to transform $\mathcal{A}$ so that it becomes $\mathcal{R}/E$-coherent: we build the product between $\mathcal{A}$ and $\boldsymbol{MN}(E)$. We recall the definition of a product automaton and we show that the product is $\mathcal{R}/E$-coherent.

▶ **Definition 27** (Product automaton [6]). *Let $\mathcal{A} = (\mathcal{F}, Q, Q_F, \Delta_\mathcal{A})$ and $\mathcal{B} = (\mathcal{F}, P, P_F, \Delta_\mathcal{B})$ be automata. The product of $\mathcal{A}$ and $\mathcal{B}$ is $\mathcal{A} \times \mathcal{B} = (\mathcal{F}, Q \times P, Q_F \times P_F, \Delta)$ where $\Delta = \{f((q_1, p_1), \ldots, (q_k, p_k)) \to (q', p') \mid f(q_1, \ldots, q_k) \to q' \in \Delta_\mathcal{A} \text{ and } f(p_1, \ldots, p_k) \to p' \in \Delta_\mathcal{B}\}$.*

▶ **Theorem 28** (Generalized Upper Bound). *Let $\mathcal{R}$ be a left-linear TRS, $\mathcal{A}$ an epsilon-free automaton, and E a set of ground equations such that $\mathcal{T}(\mathcal{F})/_{=_E}$ is finite. If $\mathcal{B} = \boldsymbol{MN}(E)$ and $\mathbb{A} = \mathcal{A} \times \mathcal{B}$ then for any $i \in \mathbb{N}$: $\mathcal{L}(\mathbb{A}_{\mathcal{R},E}^i) \subseteq \mathcal{R}_E^*(\mathcal{L}(\mathcal{A}))$.*
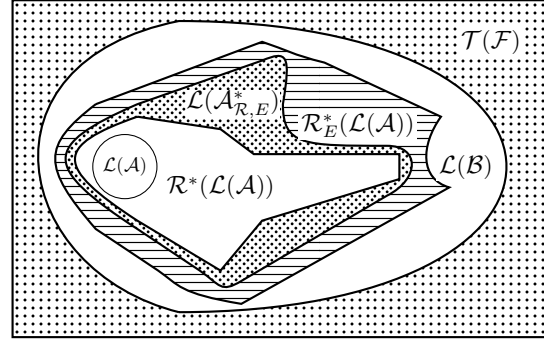
■ **Figure 4** The Generalized Upper Bound theorem (precision of completion)

Proof. Since $\mathcal{L}(\mathbb{A}) = \mathcal{L}(\mathcal{A} \times \mathcal{B}) = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$ and $\mathcal{L}(\mathcal{B}) = \mathcal{T}(\mathcal{F})$, we get that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathbb{A})$. Since both $\mathcal{A}$ and $\mathcal{B}$ are epsilon-free, so is $\mathbb{A}$. Thus, to prove $\mathcal{R}/E$-coherence of $\mathbb{A}$, we only have to prove that for all states q of $\mathbb{A}$ and for all two terms $s, t \in \mathcal{T}(\mathcal{F})$ such that (1) $s \rightarrow_{\mathbb{A}}^{\varepsilon\!\!\!/ *} q$ and (2) $t \rightarrow_{\mathbb{A}}^{\varepsilon\!\!\!/ *} q$ then $s =_E t$. Since $\mathbb{A}$ is a product automaton, q is a pair of the form (q_1, q_2) where $q_1 \in \mathcal{A}$ and $q_2 \in \mathcal{B}$. From (1) and (2) we can deduce that $s \rightarrow_{\mathcal{B}}^{\varepsilon\!\!\!/ *} q_2$ and $t \rightarrow_{\mathcal{B}}^{\varepsilon\!\!\!/ *} q_2$. Then, using Lemma 12, we get $s =_E t$. Thus $\mathbb{A}$ is $\mathcal{R}/E$-coherent and from Theorem 8, we get that $\mathcal{L}(\mathbb{A}_{\mathcal{R},E}^i) \subseteq \mathcal{R}_E^*(\mathcal{L}(\mathbb{A}))$ and $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathbb{A})$ ends the proof. ◀

▶ **Example 29.** Starting from Example 26, we can build the product between $\mathcal{A}$ and the automaton $\mathcal{B}$ found in Example 13. In $\mathcal{A} \times \mathcal{B}$, a and c are recognized by two different states, avoiding the $\mathcal{R}/E$-coherence problem of Example 26. The $\varepsilon\!\!\!/$-reduced product $\mathbb{A} = \mathcal{A} \times \mathcal{B}$ (where product states are renamed) is the automaton with $\mathcal{Q}_f = \{q_2\}$ and $\Delta = \{c \rightarrow q_0, a \rightarrow q_1, f(q_0) \rightarrow q_2, f(q_1) \rightarrow q_2\}$. Running Timbuk on $\mathbb{A}$, $\mathcal{R}$, and $E_{\mathcal{B}}$, we obtain $\mathbb{A}_{\mathcal{R},E}^*$ whose precision is now bounded by $\mathcal{R}_{E_{\mathcal{B}}}^*(\mathcal{L}(\mathcal{A}))$ and does not recognize c in a final state:

States q0 q1 q2 Final States q0 Transitions a->q1 f(q0)->q0 f(q1)->q0 f(q2)->q0 a->q0 c->q2

This provides hints to define equations for completion: we can start from an automaton $\mathcal{B}$ defining a rough approximation of the target language and build $E = \mathbf{A2E}(\mathcal{B})$. Then, we complete $\mathbb{A} = \mathcal{A} \times \mathcal{B}$ with $\mathcal{R}$ and E and obtain a tree automaton $\mathbb{A}_{\mathcal{R},E}^*$ whose precision is better or equal to $\mathcal{B}$. The set $\mathcal{R}_E^*(\mathcal{L}(\mathcal{A}))$ acts as a safeguard for completion (see Figure 4). In particular, terms of $\mathcal{R}_E^*(\mathcal{L}(\mathcal{A}))$ may not belong to $\mathcal{L}(\mathbb{A}_{\mathcal{R},E}^*)$. This is the case in Example 29, where the term b belongs to $\mathcal{R}_{E_{\mathcal{B}}}^*(\mathcal{L}(\mathcal{A}))$ but not to $\mathcal{L}(\mathbb{A}_{\mathcal{R},E}^*)$. In practice, we still need to know if E always exists (Section 7) and to generate a satisfactory E (Section 8).

7 Completeness Theorems

In this section, we prove two completeness theorems on completion. The first theorem states that if the set of reachable terms can be over-approximated by a regular language $\mathscr{L}$, then we can find a language containing reachable terms and under-approximating $\mathscr{L}$ using equational completion. The second theorem states that if the set of reachable terms is regular then completion can build it. Since the upper-bound of completion depends on $\mathcal{R}_E^*$, we first need a lemma showing that if E is built from $\mathscr{L}$ then $\mathcal{R}_E^*$ is upper-bounded by $\mathscr{L}$.

▶ **Lemma 30.** *Let $\mathcal{R}$ be a TRS over $\mathcal{F}$, $S \subseteq \mathcal{T}(\mathcal{F})$, and $\mathcal{B}$ an **RDFC** automaton such that $\mathcal{L}(\mathcal{B}) \supseteq \mathcal{R}^*(S)$ and $\mathcal{L}(\mathcal{B})$ is $\mathcal{R}$-closed. If $E_{\mathcal{B}} = \mathbf{A2E}(\mathcal{B})$ then $\mathcal{R}_{E_{\mathcal{B}}}^*(S) \subseteq \mathcal{L}(\mathcal{B})$.*

Example 31 shows that the $\mathcal{R}$-closed assumption on $\mathscr{L}$ is necessary for the lemma to hold.

▶ **Example 31.** Let $\mathcal{F} = \{a : 0, b : 0, c : 0, d : 0\}$, $S = \{a\}$, $\mathcal{R} = \{a \rightarrow b, c \rightarrow d\}$, and $\mathscr{L} = \{a, b, c\}$ where $\mathscr{L} \supseteq \mathcal{R}^*(S)$ but $\mathscr{L}$ is not $\mathcal{R}$-closed. A possible **RDFC** automaton $\mathcal{B}$, s.t. $\mathcal{L}(\mathcal{B}) = \mathscr{L}$, has a unique final state q and transitions $\{a \rightarrow q, b \rightarrow q, c \rightarrow q\}$. Thus $E_\mathcal{B} = \mathbf{A2E}(\mathcal{B})$ includes the equation $b = c$. Finally $\mathcal{R}^*_{E_\mathcal{B}}(S) = \{a, b, c, d\} \not\subseteq \mathscr{L}$.

▶ **Theorem 32** (Completeness). *Let $\mathcal{A}$ be a reduced epsilon-free tree automaton and $\mathcal{R}$ a left-linear TRS. Let $\mathcal{T}(\mathcal{F}) \supseteq \mathscr{L} \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$. If $\mathscr{L}$ is regular and $\mathcal{R}$-closed then there exists a set of ground equations E such that $\mathbb{A} = \mathcal{A} \times \mathbf{MN}(E)$, $\mathbb{A}^*_{\mathcal{R},E}$ exists and $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\mathbb{A}^*_{\mathcal{R},E}) \subseteq \mathscr{L}$.*

Proof. Since $\mathscr{L}$ is regular, we know that there exists an **RDFC** tree automaton, say $\mathcal{B}$, recognizing $\mathscr{L}$. From $\mathcal{B}$ we can infer $E_\mathcal{B} = \mathbf{A2E}(\mathcal{B})$ and then use completion to compute reachable terms. From Theorem 23, we know that completion of the automaton $\mathcal{A}$ with $\mathcal{R}$ and the set of equations $E_\mathcal{B}$ always terminates on a tree automaton $\mathcal{A}^*_{\mathcal{R},E_\mathcal{B}}$. From Theorem 8, we know that $\mathcal{L}(\mathcal{A}^*_{\mathcal{R},E_\mathcal{B}}) \subseteq \mathcal{R}^*_{E_\mathcal{B}}(\mathcal{L}(\mathcal{A}))$ provided that $\mathcal{A}$ is $\mathcal{R}/E_\mathcal{B}$-coherent. To enforce $\mathcal{R}/E_\mathcal{B}$-coherence of $\mathcal{A}$, we apply the transformation presented in Section 6. Let $\mathbb{A} = \mathcal{A} \times \mathbf{MN}(E_\mathcal{B})$. Note that since $E_\mathcal{B}$ is obtained by using the **A2E** transformation, $\mathcal{T}(\mathcal{F})/_{=_E}$ is finite (Corollary 21) and since equations of $E_\mathcal{B}$ are ground, $=_{E_\mathcal{B}}$ is decidable. The resulting automaton $\mathbb{A}$ is $\mathcal{R}/E_\mathcal{B}$-coherent. Besides, Theorem 23 also applies to $\mathbb{A}$. Thus, completion of $\mathbb{A}$ with $\mathcal{R}$ and $E_\mathcal{B}$ always ends on an automaton $\mathbb{A}^*_{\mathcal{R},E_\mathcal{B}}$. The automaton $\mathbb{A}^*_{\mathcal{R},E_\mathcal{B}}$ satisfies $\mathcal{R}^*(\mathcal{L}(\mathbb{A})) \subseteq \mathcal{L}(\mathbb{A}^*_{\mathcal{R},E_\mathcal{B}})$ (by Theorem 5) and $\mathcal{L}(\mathbb{A}^*_{\mathcal{R},E_\mathcal{B}}) \subseteq \mathcal{R}^*_{E_\mathcal{B}}(\mathcal{L}(\mathbb{A}))$ (by Theorem 28). Since $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathbb{A})$, we have $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\mathbb{A}^*_{\mathcal{R},E_\mathcal{B}})$ and $\mathcal{L}(\mathbb{A}^*_{\mathcal{R},E_\mathcal{B}}) \subseteq \mathcal{R}^*_{E_\mathcal{B}}(\mathcal{L}(\mathcal{A}))$. With Lemma 30, we get that $\mathcal{R}^*_{E_\mathcal{B}}(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\mathcal{B}) = \mathscr{L}$. ◀

In general we do not have $\mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}) \supseteq \mathscr{L}$ because $\mathcal{L}(\mathcal{A}^*_{\mathcal{R},E})$ can be more precise than $\mathscr{L}$ (See Example 29). However, this is true when $\mathscr{L} = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$, as we show in the next theorem.

▶ **Theorem 33** (Completeness for regularity preserving TRSs). *Let $\mathcal{A}$ be a reduced epsilon-free tree automaton and $\mathcal{R}$ a left-linear TRS. If $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ is regular then it is possible to compute a tree automaton recognizing $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ by equational tree automata completion.*

Proof. Let $\mathscr{L} = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$. It is $\mathcal{R}$-closed. By assumption, it is also regular. Thus, we can apply Theorem 32 to get that there exists a set of equations E and a tree automaton $\mathbb{A} = \mathcal{A} \times \mathbf{MN}(E)$ such that $\mathbb{A}^*_{\mathcal{R},E}$ exists and $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\mathbb{A}^*_{\mathcal{R},E}) \subseteq \mathscr{L}$. Since $\mathscr{L} = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$, we get $\mathcal{L}(\mathbb{A}^*_{\mathcal{R},E}) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$. ◀

Thus, completion is complete w.r.t. *all left-linear TRS classes preserving regularity.*

8   Application of the Completeness Theorem

Let us show how to take advantage of Theorem 32 to automatically verify safety properties on programs. Given an initial regular language S and a program represented by a TRS $\mathcal{R}$, we can prove that the program never reaches terms in a set *Bad* by checking that there exists a regular over-approximation $\mathscr{L} \supseteq \mathcal{R}^*(S)$ such that $\mathscr{L} \cap Bad = \emptyset$. This technique has been used to verify cryptographic protocols [1], Java programs [4] and Functional Programs [10, 14]. Theorem 32 ensures that, if there exists an $\mathcal{R}$-closed regular approximation $\mathscr{L}$ such that $\mathscr{L} \cap Bad = \emptyset$, then we can build it (or under-approximate it) using completion and an appropriate set of ground equations E. To explore all the possible E, it is enough to explore $G_\mathcal{F}(k)$ with $k \in \mathbb{N}^*$.

▶ **Definition 34** (Generated Equations for $\mathcal{F}$ and $k \in \mathbb{N}^*$). Let $\mathbb{B}(k)$ be the set of all possible **RDFC** tree automata on $\mathcal{F}$ with exactly k states. The set of *generated equations* of size k is $G_\mathcal{F}(k) = \{E \mid \mathcal{B} \in \mathbb{B}(k) \text{ and } E = \mathbf{A2E}(\mathcal{B})\}$.

The semi-algorithm to prove that $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \cap Bad = \emptyset$ works as follows: (a) We start from $k = 1$, (b) we generate $G_\mathcal{F}(k)$, (c) we try completion with $\mathcal{A}$, $\mathcal{R}$ and all $E \in G_\mathcal{F}(k)$ (completion terminates with all those E, Theorem 23). If $\mathcal{L}(\mathcal{A}_{\mathcal{R},E}^*) \cap Bad = \emptyset$ for one E, we are done. Otherwise if $\mathcal{L}(\mathcal{A}_{\mathcal{R},E}^*) \cap Bad \neq \emptyset$ for all $E \in G_\mathcal{F}(k)$, we increase k and go back to step (b). If there exists a regular over-approximation $\mathscr{L} \supseteq \mathcal{R}^*(S)$ such that $\mathscr{L} \cap Bad = \emptyset$, then this algorithm eventually reaches a tree automaton $\mathcal{B}$ such that $\mathcal{L}(\mathcal{B}) = \mathscr{L}$, $E = \mathbf{A2E}(\mathcal{B})$, and by Theorem 32, we know that $\mathcal{L}(\mathcal{A}_{\mathcal{R},E}^*) \subseteq \mathscr{L}$. Finally, since $\mathscr{L} \cap Bad = \emptyset$, we have $\mathcal{L}(\mathcal{A}_{\mathcal{R},E}^*) \cap Bad = \emptyset$.

For general TRSs, we can enumerate all equation sets of $G_\mathcal{F}(k)$ but the search space is huge. When the TRS $\mathcal{R}$ encodes a functional program, we can restrict the search space to equation sets of the form $E = E_\mathcal{R} \cup E_r \cup E_\mathcal{C}$ [10], where $E_\mathcal{R}$ and E_r are fixed and $E_\mathcal{C}$ only ranges over $\mathrm{IRR}(\mathcal{R})$. If program's functions are complete and terminating, $\mathrm{IRR}(\mathcal{R})$ is the set of constructor terms, i.e., terms containing no function call. The set $\mathcal{F}$ can be separated into a set of *defined symbols* $\mathcal{D} = \{f \mid \exists l \to r \in \mathcal{R} \text{ s.t. } \mathscr{R}oot(l) = f\}$ and *constructor symbols* $\mathcal{C} = \mathcal{F} \setminus \mathcal{D}$.

▶ **Definition 35** (E_r). For an alphabet $\mathcal{F}$, $E_r = \{f(x_1, \ldots, x_n) = f(x_1, \ldots, x_n) \mid f \in \mathcal{F}, \text{ and arity of } f \text{ is } n\}$, where $x_1 \ldots x_n$ are pairwise distinct variables.

▶ **Definition 36** ($E_\mathcal{R}$). Let $\mathcal{R}$ be a TRS, the set of $\mathcal{R}$-equations is $E_\mathcal{R} = \{l = r \mid l \to r \in \mathcal{R}\}$.

▶ **Definition 37** ($E_\mathcal{C}$ contracting equations for $\mathcal{T}(\mathcal{C})$). A set of equations is contracting for $\mathcal{T}(\mathcal{C})$, denoted by $E_\mathcal{C}$, if all equations of $E_\mathcal{C}$ are of the form $u = u|_p$ with $u \in \mathcal{T}(\mathcal{C})$, $p \neq \lambda$, $\overrightarrow{E_\mathcal{C}} = \{u \to u|_p \mid u = u|_p \in E_\mathcal{C}\}$, and $\mathrm{IRR}(\overrightarrow{E_\mathcal{C}})$ (terms of $\mathcal{T}(\mathcal{C})$ irreducible by $\overrightarrow{E_\mathcal{C}}$) is finite.

Completion is terminating if $E = E_\mathcal{R} \cup E_r \cup E_\mathcal{C}$ and $\mathcal{R}$ encodes a functional program which is terminating, complete, and is either first order [10] or higher-order [14]. Now, our objective is to define a completeness theorem for TRSs encoding those programs. Since E contains $E_\mathcal{R}$, all completed automata $\mathcal{A}_{\mathcal{R},E}^*$ will be $\mathcal{R}$-closed because $s \to_{\mathcal{A}_{\mathcal{R},E}^*}^* q$, $s \to_\mathcal{R} t$, $t \to_{\mathcal{A}_{\mathcal{R},E}^*}^* q'$ implies that $s =_{E_\mathcal{R}} t$ and $q = q'$ ($\mathcal{A}_{\mathcal{R},E}^*$ is simplified w.r.t. $E \supseteq E_\mathcal{R}$). Thus, the completeness theorem says that if there exists an $\mathcal{R}$-closed automaton $\mathcal{B}$ s.t. $\mathcal{L}(\mathcal{B}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ then there exists $E_\mathcal{C}$ such that $E = E_\mathcal{R} \cup E_r \cup E_\mathcal{C}$ and $\mathcal{L}(\mathcal{A}_{\mathcal{R},E}^*) \subseteq \mathcal{L}(\mathcal{B})$. To prove such a theorem, we need to explain how to construct a satisfying $E_\mathcal{C}$ from $\mathcal{B}$. We propose to project $\mathcal{B}$ on $\mathcal{C}$ (denoted by $\mathcal{B}/\mathcal{C}$), produce equations from $\mathcal{B}/\mathcal{C}$ with $\mathbf{A2E}$, and finally filter out all equations that are not of the form $u = u|_p$ (this is function ct).

▶ **Definition 38** (Automaton projection on $\mathcal{C}$). Let $\mathcal{B} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be an epsilon free tree automaton. The automaton $\mathcal{B}/\mathcal{C}$ is the tree automaton $\langle \mathcal{C}, \mathcal{Q}_\mathcal{C}, \mathcal{Q}_f \cap \mathcal{Q}_\mathcal{C}, \Delta_\mathcal{C} \rangle$ where $\Delta_\mathcal{C} = \{s \to q \mid s \to q \in \Delta \wedge \mathscr{R}oot(s) \in \mathcal{C}\}$ and $\mathcal{Q}_\mathcal{C}$ is the set of states occurring in the right-hand side of transitions of $\Delta_\mathcal{C}$.

Note that $\mathcal{L}(\mathcal{B}/\mathcal{C}) = \mathcal{L}(\mathcal{B}) \cap \mathcal{T}(\mathcal{C})$ and if $\mathcal{B}$ is **RDFC** so is $\mathcal{B}/\mathcal{C}$. In particular, if $\mathcal{B}$ is complete for $\mathcal{F}$, $\mathcal{B}/\mathcal{C}$ is complete for $\mathcal{C}$.

▶ **Definition 39.** Given a set of equations E, $ct(E) = \{l = r \in E \mid r = l|_p \text{ and } p \neq \lambda\}$.

In the following, we show that $E = ct(\mathbf{A2E}(\mathcal{B}))$ is a contracting set of equations, provided that $\mathcal{B}$ is **RDFC**. In particular, we show that $\mathrm{IRR}(\overrightarrow{E})$ is finite.

▶ **Lemma 40.** *If $\mathcal{B}$ is an **RDFC** automaton on $\mathcal{C}$ and $E = ct(\mathbf{A2E}(\mathcal{B}))$, then $\mathrm{IRR}(\overrightarrow{E})$ is finite and E is contracting for $\mathcal{T}(\mathcal{C})$.*

The above lemma states that $ct(\mathbf{A2E}(\mathcal{B}))$ is contracting for $\mathcal{T}(\mathcal{C})$. To have a finite set of equivalence classes on $\mathcal{T}(\mathcal{F})$ (and a terminating completion) we use $E = E_{\mathcal{R}} \cup E_r \cup E_{\mathcal{C}}$ where $E_{\mathcal{C}} = ct(\mathbf{A2E}(\mathcal{B}/\mathcal{C}))$. Now we prove that, w.r.t. approximations, E is as precise as $E_{\mathcal{B}}$.

▶ **Lemma 41.** *For a TRS $\mathcal{R}$ and an automaton $\mathcal{B}$ on $\mathcal{F}$, if $\mathcal{B}$ is **RDFC** and $\mathcal{R}$-closed and $E_{\mathcal{B}} = \mathbf{A2E}(\mathcal{B})$, $E_{\mathcal{C}} = ct(\mathbf{A2E}(\mathcal{B}/\mathcal{C}))$, and $E = E_{\mathcal{R}} \cup E_r \cup E_{\mathcal{C}}$ then $=_E \subseteq =_{E_{\mathcal{B}}}$.*

▶ **Theorem 42** ($E_{\mathcal{R}} \cup E_r \cup E_{\mathcal{C}}$ covers all $\mathcal{R}$-closed approximation automata)**.** *Let $\mathcal{R}$ be a left-linear TRS and $\mathcal{A}$ a reduced and epsilon-free tree automaton on $\mathcal{F}$. Let $\mathcal{B}$ be an $\mathcal{R}$-closed **RDFC** tree automaton such that $\mathcal{L}(\mathcal{B}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$. Let $E_{\mathcal{C}} = ct(\mathbf{A2E}(\mathcal{B}/\mathcal{C}))$, $E = E_{\mathcal{C}} \cup E_{\mathcal{R}} \cup E_r$, and $\mathbb{A} = \mathcal{A} \times \mathbf{MN}(E)$. If $\mathbb{A}_{\mathcal{R},E}^*$ exists then $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\mathbb{A}_{\mathcal{R},E}^*) \subseteq \mathcal{L}(\mathcal{B})$.*

Proof. The fact that $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\mathbb{A}_{\mathcal{R},E}^*)$ is ensured by Theorem 5. Using the Generalized Upper Bound theorem (Theorem 28), we deduce that (1) $\mathcal{L}(\mathbb{A}_{\mathcal{R},E}^*) \subseteq \mathcal{R}_E^*(\mathcal{L}(\mathcal{A}))$. From Lemma 41, we know that $=_E \subseteq =_{E_{\mathcal{B}}}$ and thus that (2) $\mathcal{R}_E^*(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{R}_{E_{\mathcal{B}}}^*(\mathcal{L}(\mathcal{A}))$. Besides, since $\mathcal{B}$ is $\mathcal{R}$-closed, $\mathcal{L}(\mathcal{B})$ is $\mathcal{R}$-closed and we can use Lemma 30 to get that (3) $\mathcal{R}_{E_{\mathcal{B}}}^*(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\mathcal{B})$. Finally, using transitivity of $\subseteq$ on (1), (2) and (3) we get $\mathcal{L}(\mathbb{A}_{\mathcal{R},E}^*) \subseteq \mathcal{L}(\mathcal{B})$.   ◀

Note that, for functional programs classes of [10] and [14], since $E_{\mathcal{C}} = ct(\mathbf{A2E}(\mathcal{B}/\mathcal{C}))$ is contracting (Lemma 40), $\mathbb{A}_{\mathcal{R},E}^*$ always exists. Thus, if there exists an $\mathcal{R}$-closed tree automaton $\mathcal{B}$ such that $\mathcal{L}(\mathcal{B}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ and $\mathcal{L}(\mathcal{B}) \cap Bad = \emptyset$, it is enough to enumerate all possible $E = E_{\mathcal{R}} \cup E_r \cup E_{\mathcal{C}}$ to find it. Since $E_{\mathcal{R}}$ and E_r are fixed, it is enough to enumerate all possible $E_{\mathcal{C}}$ on $\mathcal{C}$ using Definition 37 and the algorithm of Definition 34 (generating on $\mathcal{C}$).

▶ **Example 43.** Let $\mathcal{C} = \{0 : 0, s : 1\}$. For $k = 1$, there is only one **RDFC** automaton with 1 state. Its transitions are $\{s(q_0) \to q_0, 0 \to q_0\}$. Thus, $G_{\mathcal{C}}(1) = \{\{s(0) = 0\}\}$. For $k = 2$ there are 2 **RDFC** automata : one with transitions $\{0 \to q_0, s(q_0) \to q_1, s(q_1) \to q_1\}$ and the other with transitions $\{0 \to q_0, s(q_0) \to q_1, s(q_1) \to q_0\}$. Thus, $G_{\mathcal{C}}(2) = \{\{s(s(0)) = s(0)\}, \{s(s(0)) = 0, s(s(s(0))) = s(0)\}\}$.

We implemented this in Timbuk and used it to verify more than 20 safety properties of several first-order and higher-order functions on lists, ordered lists, trees and ordered trees. Higher-order properties include state-of-the-art examples from [21, 19, 14]. In [14], contracting equations of $E_{\mathcal{C}}$ contain variables and are generated from test sets. Here, we generate ground contracting equations $E_{\mathcal{C}}$ as shown above and use $E = E_{\mathcal{R}} \cup E_r \cup E_{\mathcal{C}}$ for completion. We transform the initial automaton $\mathcal{A}$ into $\mathbb{A}$ as in Theorem 28. The approximation is, thus, upper-bounded by $\mathcal{R}_E^*$ and we can benefit from the coverage guarantee of Theorem 42. On examples taken from [21, 19], we managed to do the same proofs with comparable execution times. On all the examples of [14], we do the same proofs (or find the counter-examples, see [14]), but in a much faster way. Appendix A presents a summary of those experiments and full details are here: `http://people.irisa.fr/Thomas.Genet/timbuk/funExperiments/`. For each example, we provide the specifications, Timbuk output, and the full result with completed automaton and generated equations in a Coq checkable file `comp.res`.

9   Conclusion and perspectives

Tree automata completion is known to cover many TRS classes preserving regularity [8, 10]. For some other classes, the question was still open. We established that, for all those classes (including those not known yet), given $\mathcal{A}$ and $\mathcal{R}$, there exists a set of equations E such that $\mathcal{A}_{\mathcal{R},E}^*$ recognizes $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$. We proved a similar theorem for the approximated case. The proofs are not constructive but give hints to enumerate sets of equations E. Finally, we showed that if a regular approximation satisfying a given property exists, we can find it by

enumerating the sets E and running completion. From an algorithmic point of view and *in the general case* (where $\mathcal{T}(\mathcal{F})/_{=E}$ is finite), since we enumerate tree automata $\mathcal{B}$ on $\mathcal{T}(\mathcal{F})$ to generate sets of equations E, we could directly take advantage of $\mathcal{B}$ to perform automata simplification and thus replace equations.

However, equations are strictly more powerful than tree automata to define approximations. This can be observed on functional programs (Section 8) where $\mathcal{T}(\mathcal{C})/_{=E_\mathcal{C}}$ is finite (and $E_\mathcal{C}$ is generated using a tree automaton) but $\mathcal{T}(\mathcal{F})/_{=E}$ is not [14] and E cannot be defined with an automaton. On functional programs, Theorem 42 shows that enumeration can be restricted to sets of ground contracting equations on constructor symbols. This makes enumeration efficient enough to automatically verify properties on first-order and higher programs. Experiments shows that this approach tackles state-of-the-art automatic verification problems for first-order and higher-order programs. The completeness Theorem for functional programs ranges over $\mathcal{R}$-closed **RDFC** approximation automata. However, there exist $\mathcal{R}$-closed approximations that are not recognized by $\mathcal{R}$-closed **RDFC** tree automata.

▶ **Example 44.** Let $\mathcal{F} = \{f : 1, a : 0, b : 0\}$, $\mathcal{R} = \{a \to b\}$ and $\mathscr{L} = \{f(b), a, b\}$. The language $\mathscr{L}$ is $\mathcal{R}$-closed and regular. There exists no $\mathcal{R}$-closed **RDFC** tree automaton recognizing $\mathscr{L}$. In any $\mathcal{R}$-closed **RDFC** tree automaton, a and b need to be recognized by the same state, say q, and thus $f(b)$ needs to be recognized using a transition $f(q) \to q_f$ where q_f is final. Thus, this automaton recognizes $f(a)$ which does not belong to $\mathscr{L}$.

Such approximations are thus out of the scope of Theorem 42, and cannot be found by enumerating $E_\mathcal{C}$, because E contains $E_\mathcal{R}$ and the completed automata are $\mathcal{R}$-closed. However, the above approximation is in the scope of Theorem 32. We think that it is possible to explore the set of all possible equation sets using $E = E_r \cup E_\mathcal{F}$ where $E_\mathcal{F}$ is contracting on $\mathcal{T}(\mathcal{F})$ and to prune the search space using Counter Example Guided Abstraction Refinement like [19]. This would permit to have an efficient equation generation for general TRSs and widen its applicability to non-terminating functional programs, cryptographic protocols, etc.

A last perspective is to extend those results to non-left-linear TRSs. Dealing with regular languages and non-left-linear rules is known to be more challenging than the left-linear case [24, 3, 7]. Nevertheless, there could be a nice surprise here. For non-left-linear TRSs, completion is known to be sound and precise as long as the completed tree automaton is kept deterministic [8]. Completion itself does not preserve determinism but, in Section 8, all the completed automata of the experiments are deterministic. This is a consequence of the fact that E contains E_r (makes the automaton ϵ-deterministic) and $E_\mathcal{R}$ (merges all states related by an ϵ-transition). Thus, when using $E = E_\mathcal{R} \cup E_r \cup E_\mathcal{C}$, completion may build precise over-approximations for non-left-linear TRSs as it does for left-linear ones.

―― **References** ――――――――――――――――――――――――――――――――――

1 A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P.-C. Héam, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santos Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA Tool for the automated validation of internet security protocols and applications. In *CAV'2005*, volume 3576 of *LNCS*, pages 281–285. Springer, 2005.

2 F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

3 Y. Boichut, R. Courbis, P.-C. Héam, and O. Kouchnarenko. Handling non left-linear rules when completing tree automata. *IJFCS*, 20(5), 2009.

4   Y. Boichut, T. Genet, T. Jensen, and L. Leroux. Rewriting Approximations for Fast Prototyping of Static Analyzers. In *RTA'07*, volume 4533 of *LNCS*, pages 48–62. Springer, 2007.

5   Y. Boichut, P.-C. Héam, and O. Kouchnarenko. Automatic Approximation for the Verification of Cryptographic Protocols. In *Proc. AVIS'2004, joint to ETAPS'04, Barcelona (Spain)*, 2004.

6   H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. Löding, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. `http://tata.gforge.inria.fr`, 2008.

7   B. Felgenhauer and R. Thiemann. Reachability Analysis with State-Compatible Automata. In *LATA'14*, volume 8370 of *LNCS*, pages 347–359. Springer, 2014.

8   G. Feuillade, T. Genet, and V. Viet Triem Tong. Reachability Analysis over Term Rewriting Systems. *Journal of Automated Reasonning*, 33 (3-4):341–383, 2004. URL: `http://people.irisa.fr/Thomas.Genet/publications.html`.

9   T. Genet. Decidable Approximations of Sets of Descendants and Sets of Normal Forms. In *RTA'98*, volume 1379 of *LNCS*, pages 151–165. Springer, 1998.

10   T. Genet. Termination Criteria for Tree Automata Completion. *Journal of Logical and Algebraic Methods in Programming*, 85, Issue 1, Part 1:3–33, 2016.

11   T. Genet. Automata Completion and Regularity Preservation. Technical report, INRIA, 2017. URL: `https://hal.inria.fr/hal-01501744`.

12   T. Genet, Y. Boichut, B. Boyer, T. Gillard, T. Haudebourg, and S. Lê Cong. Timbuk 3.2 – a Tree Automata Library. IRISA / Université de Rennes 1, 2017. URL: `http://people.irisa.fr/Thomas.Genet/timbuk/`.

13   T. Genet, T. Gillard, T. Haudebourg, and S. Lê Cong. Extending timbuk to Verify Functional Programs. In *WRLA'18*, LNCS. Springer, 2018. To be published.

14   T. Genet, T. Haudebourg, and T. Jensen. Verifying higher-order functions with tree automata. In *FoSSaCS'18*, LNCS. Springer, 2018. To be published.

15   T. Genet and R. Rusu. Equational tree automata completion. *Journal of Symbolic Computation*, 45:574–597, 2010.

16   A. Geser, D. Hofbauer, J. Waldmann, and H. Zantema. On tree automata that certify termination of left-linear term rewriting systems. In *RTA'05*, volume 3467 of *LNCS*, pages 353–367. Springer, 2005.

17   F. Jacquemard. Decidable approximations of term rewriting systems. In H. Ganzinger, editor, *Proc. of RTA'96*, volume 1103 of *LNCS*, pages 362–376. Springer, 1996.

18   Dexter Kozen. On the Myhill-Nerode theorem for trees. *Bull. Europ. Assoc. Theor. Comput. Sci.*, 47:170–173, June 1992.

19   Y. Matsumoto, N. Kobayashi, and H. Unno. Automata-Based Abstraction for Automated Verification of Higher-Order Tree-Processing Programs. In *APLAS'15*, volume 9458 of *LNCS*, pages 295–312. Springer, 2015.

20   A. Middeldorp. Approximations for strategies and termination. *ENTCS*, 70(6):1–20, 2002.

21   L. Ong and S. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *POPL'11*. ACM, 2011.

22   W. Snyder. Efficient Ground Completion: An *O(n log n)* Algorithm for Generating Reduced Sets of Ground Rewrite Rules Equivalent to a Set of Ground Equations E. In *RTA'89*, volume 355 of *LNCS*, pages 419–433. Springer, 1989.

23   T. Takai. A Verification Technique Using Term Rewriting Systems and Abstract Interpretation. In *RTA'04*, volume 3091 of *LNCS*, pages 119–133. Springer, 2004.

24   T. Takai, Y. Kaji, and H. Seki. Right-linear finite-path overlapping term rewriting systems effectively preserve recognizability. In *RTA'11*, volume 1833 of *LNCS*. Springer, 2000.

A Experiments

Timbuk Spec.	Description	P/C	Comp. Time	Eq. Gen. Time
delete	not (member A (delete A A_and_B_list))	P	0.01s	0.01s
delete2	(member B (delete A A_and_B_list))	P	0.01s	0.01s
deleteBasic	(delete A A_and_B_list) removes all occurrences of A	P	0.01s	0.01s
reverseFirstOrder	reverse [A,...A,B,...,B] does not produce lists with a A before a B	P	0.01s	0.03s
reverseFirstOrder2	invsorted (reverse [A,...A,B,...,B])	P	0.02s	0.13s
incTree	not (member 0 (increment nat_tree))	P	0.08s	1.05s
replaceTree	not (member A (replace A C A_and_B_tree))	P	0.44s	6.13s
orderedTree	ordered ordered_A_and_B_tree	P	0.16s	6.73s
insertTree	ordered (insert A_and_B_list emptyTree)	-	-	Timeout
orderedTreeTraversal	sorted (infix-traversal ordered_A_and_B_tree)	P	0.13s	1.71s
orderTreeTraversalBug	sorted (prefix-traversal ordered_A_and_B_tree)	C	0.2s	-
mapPlus	no 0 in (map (plus 1) nat_list)	P	0.02s	0.08s
filterEven	not (exists even (filter odd nat_list))	P	0.12s	1.16s
filterEvenBug	not (exists odd (filter odd nat_list))	C	0.09s	-
insertionSort	(sorted leq (sort leq A_and_B_list))	P	0.04s	0.11s
insertionSortBug	(sorted geq (sort leq A_and_B_list))	C	0.59s	-
filterNz	(forAll nz (filter nz nat_list))	P	0.01s	0.11s
mapTree	no 0 in (map (plus 1) nat_tree)	P	0.03s	16.15s
mapTree2	not (member 0 (map (plus 1) nat_tree)	-	-	Timeout
reverse	(sorted geq (reverse ordered_A_B_list))	P	0.04s	0.47s
mapSquare	(filter (eq 2) (map square nat_list)) is empty	P	0.31s	4.25s
foldRightMult	(foldRight mult nonzero_nat_list 1) is not 0	P	0.01s	0.01s
foldRightMult2	(foldRight mult nonzero_nat_list 3) is not 2	P	0.05s	0.29s
foldLeftPlus	even (foldLeft plus 0 even_nat_list)	P	0.01s	0.21s

The above table gives a summary of the experiments carried out with Timbuk. The source of the programs, trace of execution, Coq certificates, etc. can be found here: `http://people.irisa.fr/Thomas.Genet/timbuk/funExperiments/`. The 'Timbuk Spec.' column gives the name of the Timbuk specification file that was used (it is also available in Timbuk's distribution). The first 11 examples are first order programs and the 13 remaining are higher-order programs. The 'Description' column gives a short description of the property we want to prove. In the corresponding Timbuk specification this is the initial language and encoded by either a tree automaton or a *simplified regular expression* [13]. The 'P/C'

column says if Timbuk has done a (P)roof of the property or found a (C)ounter example. 'Comp. Time' stands for completion time and 'Eq. Gen. Time' for equation generation time. On some examples, the equation generation algorithm times out and completion cannot be performed.

B   Additional proofs

This section contains some the proofs of [11].

▶ **Lemma** (16). *For all **RF** tree automata $\mathcal{B}$, for all states $q \in \mathcal{B}$ there exists a natural number $k \in \mathbb{N}$ for which the set $[\![q]\!]_{\mathcal{B}}^{k}$ is a fixpoint.*

Proof. We make a proof by contradiction. Assume that one set of state representatives $[\![q]\!]_{\mathcal{B}}$ is infinite. Let $\mathcal{Q}$ be the set of states of $\mathcal{B}$ and $t \in [\![q]\!]_{\mathcal{B}}$ be a term s.t. $|t| > \mathcal{C}\mathrm{ard}(\mathcal{Q})$. Assume that we label each subterm of t by the state recognizing it in $\mathcal{B}$. Since height of t is greater than $\mathcal{C}\mathrm{ard}(\mathcal{Q})$, by the pigeonhole principle we know that there exists $q' \in \mathcal{B}$ and a path in the tree t such that q' appears at least two times. Let $p, r \in \mathscr{P}os(t)$ be the positions of the two subterms recognized by q'. By definition of state representatives, we know that $t|_p \in [\![q']\!]_{\mathcal{B}}$ and $t|_r \in [\![q']\!]_{\mathcal{B}}$. Since p and r are on the same path, we know that $t|_p$ is a strict subterm of $t|_r$ (or the opposite). This contradicts Definition 14 that forbids a term and a strict subterm to belong to the same set of representatives. ◀

▶ **Lemma** (19). *Let $\mathcal{B} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be an **RDFC** tree automaton and $E_{\mathcal{B}} = \mathbf{A2E}(\mathcal{B})$. For all $s \in \mathcal{T}(\mathcal{F})$, there exists a unique state $q \in \mathcal{Q}$ such that $s \rightarrow_{\mathcal{B}}^{*} q$ and for all state representatives $u \in [\![q]\!]_{\mathcal{B}}$, $s =_{E_{\mathcal{B}}} u$.*

Proof. We make a proof by induction on the height of s. If s is a constant, since $\mathcal{B}$ is complete and deterministic there exists a unique transition $s \rightarrow q \in \Delta$. By construction of $E_{\mathcal{B}}$, we know that there are equations with s on the left-hand side and all state representatives of $[\![q]\!]_{\mathcal{B}}$ on the right-hand side. For all equations $s = u$ with $u \in [\![q]\!]_{\mathcal{B}}$ we thus trivially have $s =_{E_{\mathcal{B}}} u$. This concludes the base case.

Now, we assume that the property is true for terms of height lesser or equal to n. Let $s = f(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms of height lesser or equal to n. Since $\mathcal{B}$ is complete, we know that there exists a state q such that $f(t_1, \ldots, t_n) \rightarrow_{\mathcal{B}}^{*} q$, i.e., there exists states $q_1, \ldots, q_n$ such that $f(q_1, \ldots, q_n) \rightarrow q \in \Delta$ and $t_i \rightarrow_{\mathcal{B}}^{*} q_i$ for $1 \leq i \leq n$. Using the induction hypothesis we get that there exist states q_i' in $\mathcal{B}$ and terms $[\![q_i']\!]_{\mathcal{B}}$ such that $t_i \rightarrow_{\mathcal{B}}^{*} q_i$ and $t_i =_{E_{\mathcal{B}}} u_i$ for $u_i \in [\![q_i']\!]_{\mathcal{B}}$ and for $1 \leq i \leq n$. Since $\mathcal{B}$ is deterministic, from $t_i \rightarrow_{\mathcal{B}}^{*} q_i$ and $t_i \rightarrow_{\mathcal{B}}^{*} q_i'$ we get that $q_i = q_i'$ and thus $t_i =_{E_{\mathcal{B}}} u_i$ for $u_i \in [\![q_i]\!]_{\mathcal{B}}$, with $1 \leq i \leq n$. Besides, since $f(q_1, \ldots, q_n) \rightarrow q \in \Delta$, we know that $E_{\mathcal{B}}$ contains the equations $f(u_1, \ldots, u_n) = u$ for all $u_i \in [\![q_i]\!]_{\mathcal{B}}$, for all $1 \leq i \leq n$ and for all $u \in [\![q]\!]_{\mathcal{B}}$. Thus q is the unique state such that $f(t_1, \ldots, t_n) \rightarrow_{\mathcal{B}}^{*} q$. Furthermore, $f(t_1, \ldots, t_n) =_{E_{\mathcal{B}}} f(u_1, \ldots, u_n) =_{E_{\mathcal{B}}} u$ for all $u_i \in [\![q_i]\!]_{\mathcal{B}}$, for all $1 \leq i \leq n$ and for all $u \in [\![q]\!]_{\mathcal{B}}$. ◀

▶ **Lemma** (20). *Let $\mathcal{B} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be an **RDFC** tree automaton and $E_{\mathcal{B}} = \mathbf{A2E}(\mathcal{B})$. For all $s, t \in \mathcal{T}(\mathcal{F})$, $s =_{E_{\mathcal{B}}} t \iff (\exists q : \{s, t\} \subseteq \mathcal{L}(\mathcal{B}, q))$.*

Proof. For s and t, using Lemma 19, we know that there exist unique states $q, q' \in \mathcal{Q}$ such that $s \rightarrow_{\mathcal{B}}^{*} q$, $t \rightarrow_{\mathcal{B}}^{*} q'$ and for all state representatives $u \in [\![q]\!]_{\mathcal{B}}$ and $v \in [\![q']\!]_{\mathcal{B}}$, we have $s =_{E_{\mathcal{B}}} u$ and $t =_{E_{\mathcal{B}}} v$. We first prove the left to right implication. From $s =_{E_{\mathcal{B}}} t$ we obtain that $u =_{E_{\mathcal{B}}} v$, where u and v are state representatives. By construction of term representatives, for all states q we know that $[\![q]\!]_{\mathcal{B}}$ only contains terms recognized by q in

$\mathcal{B}$. Since $\mathcal{B}$ is deterministic, if $q \neq q'$ then we can conclude that $[\![q]\!]_\mathcal{B} \cap [\![q']\!]_\mathcal{B} = \emptyset$. Thus, the only possibility to have $u =_{E_\mathcal{B}} v$ is to have an equation $u = v$ in $E_\mathcal{B}$. This entails that u and v belong to the same set of representatives: $[\![q]\!]_\mathcal{B} = [\![q']\!]_\mathcal{B}$, which entails that $q = q'$. Then $s \to_\mathcal{B}^* q$ and $t \to_\mathcal{B}^* q$ entails that $\{s, t\} \subseteq \mathcal{L}(\mathcal{B}, q)$. To prove the right to left implication, it is enough to point out that because of the determinism of $\mathcal{B}$ having $t \to_\mathcal{B}^* q'$ (the initial assumption) and having $t \to_\mathcal{B}^* q$ (the fact that $t \in \mathcal{L}(\mathcal{B}, q)$) is possible only if $q = q'$. This entails that u and v have a common set of representatives and thus for all representatives u of this set $s =_{E_\mathcal{B}} u =_{E_\mathcal{B}} t$. ◄

▶ **Lemma** (21). *Let $\mathcal{B} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be an **RDFC** tree automaton. If $E_\mathcal{B}$ is the set of equations inferred from $\mathcal{B}$ then $\mathcal{T}(\mathcal{F})/_{=_{E_\mathcal{B}}}$ is finite.*

Proof. Using Lemma 19, we know that for all terms $t \in \mathcal{T}(\mathcal{F})$ there exists a state $q \in \mathcal{Q}$ and a state representative $u \in [\![q]\!]_\mathcal{B}$ such that $t \to_\mathcal{B}^* q$ and $t =_{E_\mathcal{B}} u$. Since the number of states of $\mathcal{B}$ is finite, and since the set of state representatives u is finite for all states of $\mathcal{B}$ (Lemma 16), so is the number of equivalence classes of $\mathcal{T}(\mathcal{F})/_{=_{E_\mathcal{B}}}$. ◄

▶ **Lemma** (22). *Let $\mathcal{R}$ be a TRS, $\mathcal{A}$ a $\stackrel{\epsilon}{\leftarrow}$-reduced tree automaton, $\mathcal{B}$ an **RDFC** tree automaton and $E_\mathcal{B} = \boldsymbol{A2E}(\mathcal{B})$. Let $\mathcal{A}^*$ be the limit of the completion of $\mathcal{A}$ by $\mathcal{R}$ and $E_\mathcal{B}$. For all states $q \in \mathcal{A}^*$, for all terms $s \in \mathcal{T}(\mathcal{F})$ such that $s \to_{\mathcal{A}^*}^{\stackrel{\epsilon}{\leftarrow}*} q$, there exists a state $q'_\mathcal{B} \in \mathcal{B}$, a term $u \in [\![q'_\mathcal{B}]\!]_\mathcal{B}$ such that $u =_{E_\mathcal{B}} s$ and $u \to_{\mathcal{A}^*}^{\stackrel{\epsilon}{\leftarrow}*} q$.*

Proof. Note that if $\mathcal{A}$ is $\stackrel{\epsilon}{\leftarrow}$-reduced, then so is $\mathcal{A}^*$ (cf. Lemma 44 of [10]). This is easy to figure out since all states added during completion recognize at least one term with $\to_\mathcal{A}^{\stackrel{\epsilon}{\leftarrow}*}$, and this is trivially preserved by simplification. By induction on the height of s we show that the representative u exists and is recognized by q. If s is of height 1 (it is a constant) then, by construction of state representatives, we know that s is a representative. Thus $s = u \to_{\mathcal{A}^*}^{\stackrel{\epsilon}{\leftarrow}*} q$.

For the inductive case, assume that the property is true for all terms of height lesser or equal to n. Let $s = f(s_1, \ldots, s_n)$ be a term of height $n + 1$. By assumption, we know that $f(s_1, \ldots, s_n) \to_{\mathcal{A}^*}^{\stackrel{\epsilon}{\leftarrow}*} q$. From $f(s_1, \ldots, s_n) \to_{\mathcal{A}^*}^{\stackrel{\epsilon}{\leftarrow}*} q$, we obtain that there exists states $q_1, \ldots, q_n$ of $\mathcal{A}^*$ such that $s_i \to_{\mathcal{A}^*}^{\stackrel{\epsilon}{\leftarrow}*} q_i$ for $i = 1, \ldots, n$ and a transition $f(q_1, \ldots, q_n) \to q$ in $\mathcal{A}^*$. Using the induction hypothesis on q_i, $i = 1, \ldots, n$ we get that there exist state representatives u_i such that $s_i =_{E_\mathcal{B}} u_i$ and $u_i \to_{\mathcal{A}^*}^{\stackrel{\epsilon}{\leftarrow}*} q_i$ for $i = 1, \ldots, n$. Then, since $f(q_1, \ldots, q_n) \to q$ in $\mathcal{A}^*$ we know that $f(u_1, \ldots, u_n) \to_{\mathcal{A}^*}^{\stackrel{\epsilon}{\leftarrow}*} q$. If $f(u_1, \ldots, u_n)$ is a state representative we are done since $f(s_1, \ldots, s_n) =_{E_\mathcal{B}} f(u_1, \ldots, u_n)$ and $f(u_1, \ldots, u_n) \to_{\mathcal{A}^*}^{\stackrel{\epsilon}{\leftarrow}*} q$. Otherwise, by definition of state representatives, for $u = f(u_1, \ldots, u_n)$ not to belong to the representatives there is a position p in u, different from the root position such that the subterm $u|_p$ is itself a state representative and it belongs to the same class as u, i.e., $u =_{E_\mathcal{B}} u|_p$. Since $u_1, \ldots, u_n$ are state representatives and $f(u_1, \ldots, u_n)$ is in the same equivalence class as $u|_p$ which is a state representative, we know that the equation $f(u_1, \ldots, u_n) = u|_p$ necessarily belongs to $E_\mathcal{B}$. Besides, for $u \to_{\mathcal{A}^*}^{\stackrel{\epsilon}{\leftarrow}*} q$ to hold, we know that there exists a state q' such that $u[u|_p]_p \to_{\mathcal{A}^*}^{\stackrel{\epsilon}{\leftarrow}*} u[q']_p \to_{\mathcal{A}^*}^{\stackrel{\epsilon}{\leftarrow}*} q$. Thus, $f(u_1, \ldots, u_n) \to_{\mathcal{A}^*}^{\stackrel{\epsilon}{\leftarrow}*} q$ and $u|_p \to_{\mathcal{A}^*}^{\stackrel{\epsilon}{\leftarrow}*} q'$. Then, since $E_\mathcal{B}$ contains the equation $f(u_1, \ldots, u_n) = u|_p$, and since $\mathcal{A}^*$ is simplified w.r.t. $E_\mathcal{B}$, we necessarily have $q = q'$ in $\mathcal{A}^*$. Finally, we have $f(s_1, \ldots, s_n) =_{E_\mathcal{B}} f(u_1, \ldots, u_n) =_{E_\mathcal{B}} u|_p$ and $u|_p \to_{\mathcal{A}^*}^{\stackrel{\epsilon}{\leftarrow}*} q$ where $u|_p$ is a state representative. ◄

▶ **Theorem** (23). *Let $\mathcal{R}$ be a TRS, $\mathcal{A}$ a $\stackrel{\epsilon}{\leftarrow}$-reduced tree automaton, $\mathcal{B}$ be an **RDFC** tree automaton and $E_\mathcal{B} = \boldsymbol{A2E}(\mathcal{B})$. Let n be the number of all states representatives of $\mathcal{B}$. The automaton $\mathcal{A}^*$, limit of the completion of $\mathcal{A}$ with $\mathcal{R}$ and $E_\mathcal{B}$, has n states or less.*

Proof. Recall that the number n of state representatives is finite (cf. Lemma 16). Assume that $\mathcal{A}^*$ has m distinct states with $m > n$. From Lemma 22 we know that for all states $q \in \mathcal{A}^*$, there exists a state representative u such that $u \to_{\mathcal{A}^*}^{\wr *} q$. Since there are only n state representatives, by pigeon hole principle, we know that there is necessarily one state representative u recognized by two distinct states q_1 and q_2 of $\mathcal{A}^*$. Thus, $u \to_{\mathcal{A}^*}^{\wr *} q_1$ and $u \to_{\mathcal{A}^*}^{\wr *} q_2$. Besides, by construction of $E_{\mathcal{B}}$, we know that the equation $u = u$ is part of $E_{\mathcal{B}}$. This contradicts the fact that $\mathcal{A}^*$ is simplified w.r.t. $E_{\mathcal{B}}$.   ◀

▶ **Lemma** (30). *Let $\mathcal{R}$ be a TRS over $\mathcal{F}$, $S \subseteq \mathcal{T}(\mathcal{F})$, and $\mathcal{B}$ an **RDFC** automaton such that $\mathcal{L}(\mathcal{B}) \supseteq \mathcal{R}^*(S)$ and $\mathcal{L}(\mathcal{B})$ is $\mathcal{R}$-closed. If $E_{\mathcal{B}} = \boldsymbol{A2E}(\mathcal{B})$ then $\mathcal{R}_{E_{\mathcal{B}}}^*(S) \subseteq \mathcal{L}(\mathcal{B})$.*

Proof. We prove that for all natural number $k >= 0$, if $s \in S$ and $s \to_{\mathcal{R}/E_{\mathcal{B}}}^k t$ then $t \in \mathcal{L}(\mathcal{B})$ where $\to_{\mathcal{R}/E_{\mathcal{B}}}^k$ denotes k steps of rewriting by $\mathcal{R}$ modulo $E_{\mathcal{B}}$. By induction on k. If $k = 0$ then $s =_{E_{\mathcal{B}}} t$. Using Lemma 20 on $s =_{E_{\mathcal{B}}} t$, we get that there exists a state q of $\mathcal{B}$ such that $s \to_{\mathcal{B}}^* q$ and $t \to_{\mathcal{B}}^* q$. Since $s \in S$ and $S \subseteq \mathcal{L}(\mathcal{B})$ there exists a final state q_f of $\mathcal{B}$ such that $s \to_{\mathcal{B}}^* q_f$. Since $\mathcal{B}$ is deterministic we obtain that $q = q_f$. Thus t is recognized by $\mathcal{B}$. For the inductive case, we assume that the property is true for a given k and we show that it is true for $k + 1$. Let $s \to_{\mathcal{R}/E}^{k+1} t$, i.e., we have terms s', s'', and t' such that $s \to_{\mathcal{R}/E}^k s' =_{E_{\mathcal{B}}} s'' \to_{\mathcal{R}} t' =_{E_{\mathcal{B}}} t$. Using the induction hypothesis, we get that s' is recognized by $\mathcal{B}$. Since $\mathcal{L}(\mathcal{B})$ is $\mathcal{R}$-closed, we know that t' is also recognized by $\mathcal{B}$. Thus, there exists a final state q_f such that $t' \to_{\mathcal{B}}^* q_f$. Finally, as above, applying Lemma 20 on the fact that $t' \to_{\mathcal{B}}^* q_f$ and $t' =_{E_{\mathcal{B}}} t$ gives us that $t \to_{\mathcal{B}}^* q_f$.   ◀

A Diagrammatic Axiomatisation of Fermionic Quantum Circuits

Amar Hadzihasanovic
RIMS, Kyoto University, Japan
ahadziha@kurims.kyoto-u.ac.jp

Giovanni de Felice
Department of Computer Science, University of Oxford, United Kingdom
giovanni.defelice@cs.ox.ac.uk

Kang Feng Ng
Department of Computer Science, University of Oxford, United Kingdom
kangfeng.ng@cs.ox.ac.uk

──────── **Abstract** ────────

We introduce the fermionic ZW calculus, a string-diagrammatic language for fermionic quantum computing (FQC). After defining a fermionic circuit model, we present the basic components of the calculus, together with their interpretation, and show how the main physical gates of interest in FQC can be represented in the language. We then list our axioms, and derive some additional equations. We prove that the axioms provide a complete equational axiomatisation of the monoidal category whose objects are quantum systems of finitely many local fermionic modes, with operations that preserve or reverse the parity (number of particles mod 2) of states, and the tensor product, corresponding to the composition of two systems, as monoidal product. We achieve this through a procedure that rewrites any diagram in a normal form. We conclude by showing, as an example, how the statistics of a fermionic Mach-Zehnder interferometer can be calculated in the diagrammatic language.

1 Introduction

The ZW calculus is a string-diagrammatic language for qubit quantum computing, introduced by the first author in [16]. Developing ideas of Coecke and Kissinger [6], it refined and extended the earlier ZX calculus [4, 1], while keeping some of its most convenient properties, such as the ability to handle diagrams as undirected labelled multigraphs. In the version of [17, Chapter 5], it provided the first complete equational axiomatisation of the monoidal category of qubits and linear maps, with the tensor product as monoidal product. Soon after its publication, the third author and Q. Wang derived from it a universal completion of the ZX calculus [28, 18].

Since its early versions, the ZX calculus has had the advantage of including familiar gates from the circuit model of quantum computing [29, Chapter 4], such as the Hadamard gate and the CNOT gate, either as basic components of the language, or as simple composite diagrams. This facilitates the transition between formalisms and the application to known algorithms and protocols, and is related to the presence of a simple, well-behaved "core" of the ZX calculus, modelling the interaction of two strongly complementary observables [5], in the guise of special commutative Frobenius algebras [9]. Access to complementary observables is fundamental in quantum computing schemes such as the one-way quantum computer, to which the ZX calculus was applied in [11].

The ZW calculus only includes one special commutative Frobenius algebra, corresponding to the computational basis, as a basic component. On the other hand, as noted already in [16], the ZW calculus has a fundamentally different "core", which is obtained by removing a single component that does not interact as naturally with the rest. This core has the property of only representing maps that have a definite parity with respect to the computational basis: the subspaces spanned by basis states with an even or odd number of 1s are either preserved, or interchanged by a map. This happens to be compatible with an interpretation of the basis states of a single qubit as the empty and occupied states of a *local fermionic mode*, the unit of information of the *fermionic quantum computing* (FQC) model.

Fermionic quantum computing is computationally equivalent to qubit computing [3]. The connection with the ZW calculus suggested that an independent fermionic version of the calculus could be developed, combining the best of both worlds with respect to FQC rather than qubit computing: the superior structural properties of the ZW calculus, including an intuitive normalisation procedure for diagrams, together with the superior hands-on features of the ZX calculus.

In this paper, we present such an axiomatisation, to which we refer as the *fermionic ZW calculus*. We start by defining our model in Section 2: the monoidal category **LFM** of local fermionic modes and maps that either preserve or reverse the parity of a state, with the tensor product of $\mathbb{Z}_2$-graded Hilbert spaces as the monoidal product. We introduce a number of physical gates from which one may build fermionic quantum circuits: the beam splitter, the phase gates, the fermionic swap gate, and the empty and occupied state preparations. Finally, we describe our diagrammatic language with its interpretation in **LFM**, and show that all the physical gates have simple diagrammatic representations.

In Section 3, we list the axioms of the fermionic ZW calculus, and state several derived equations, whose proofs are appended at the end of the paper. We introduce short-hand notation for certain composite diagrams (sometimes called the "spider" notation in categorical quantum mechanics [7, Section 8.2]), and prove inductive generalisations of the axioms. Then, in Section 4, we prove our main theorem, that the fermionic ZW calculus is an axiomatisation of **LFM**. We achieve this by defining a normal form for diagrams, from which one can easily read the interpretation in **LFM**, and showing that any diagram can be rewritten in normal form using the axioms.

Finally, in Section 5, as a first practical example, we calculate in the diagrammatic language the statistics of a simple circuit, the fermionic Mach-Zehnder interferometer.

2   The model and the components

The basic systems in FQC are *local fermionic modes* (LFMs), physical sites that are either empty or occupied by a single spinless fermionic particle [3]. We indicate the empty and occupied states of a LFM as $|\,0\,\rangle$ and $|\,1\,\rangle$, respectively, in bra-ket notation.

Much like the computational basis states of a qubit, we can see these as an orthonormal basis for the two-dimensional complex Hilbert space B. We note that the "naive" translation from LFMs to qubits does not preserve entanglement and locality properties [15, 10]; see however [13].

States of a composite system of n LFMs correspond to states of the n-fold tensor product $B^{\otimes n}$. However, not all physical states or operations on qubits are accessible as physical states or operations on LFMs. The Hilbert space of a system of n LFMs splits as $H_0 \oplus H_1$, where H_0 is spanned by states where an even number of LFMs is occupied, and H_1 by states where an odd number of LFMs is occupied. Then, any physical operation $f : H_0 \oplus H_1 \to K_0 \oplus K_1$ must either preserve, or invert the parity, that is, either map H_0 to K_0 and H_1 to K_1, or map H_0 to K_1 and H_1 to K_0. This is called the *parity superselection rule*; see [2, 10] for a discussion.

These operations assemble into a category, as follows.

▶ **Definition 1.** A $\mathbb{Z}_2$-*graded Hilbert space* is a complex Hilbert space H decomposed as a direct sum $H_0 \oplus H_1$. A *pure map* $f : H \to K$ of $\mathbb{Z}_2$-graded Hilbert spaces is a bounded linear map $f : H \to K$ such that $f(H_0) \subseteq K_0$ and $f(H_1) \subseteq K_1$ (*even* map), or $f(H_0) \subseteq K_1$ and $f(H_1) \subseteq K_0$ (*odd* map).

Given two $\mathbb{Z}_2$-graded Hilbert spaces H, K, the tensor product $H \otimes K$ can be decomposed as $(H \otimes K)_0 := (H_0 \otimes K_0) \oplus (H_1 \otimes K_1)$, and $(H \otimes K)_1 := (H_0 \otimes K_1) \oplus (H_1 \otimes K_0)$. Then, the tensor product (as maps of Hilbert spaces) of a pair of pure maps $f : H \to K$, $f' : H' \to K'$ is a pure map $f \otimes f' : H \otimes H' \to K \otimes K'$ of $\mathbb{Z}_2$-graded Hilbert spaces. The $\mathbb{Z}_2$-graded Hilbert space $\mathbb{C} \oplus 0$ acts as a unit for the tensor product.

We write $\mathbf{Hilb}^{\mathbb{Z}_2}$ for the symmetric monoidal category of $\mathbb{Z}_2$-graded Hilbert spaces and pure maps, with the tensor product as monoidal product.

▶ **Remark 2.** The zero maps $0 : H \to K$ are the only pure maps between two $\mathbb{Z}_2$-graded Hilbert spaces H, K that are both even and odd.

▶ **Definition 3.** We write $\mathbf{LFM}$ for the full monoidal subcategory of $\mathbf{Hilb}^{\mathbb{Z}_2}$ whose objects are n-fold tensor products of $B := \mathbb{C} \oplus \mathbb{C}$, for all $n \in \mathbb{N}$.

Here, B_0 is the span of $|0\rangle$, and B_1 is the span of $|1\rangle$. As customary, we write $|b_1 \dots b_n\rangle$ for the basis state $|b_1\rangle \otimes \dots \otimes |b_n\rangle$ of $B^{\otimes n}$, where $b_i \in \{0, 1\}$, for $i = 1, \dots, n$.

The category $\mathbf{LFM}$ admits, in fact, the structure of a dagger compact closed category in the sense of [31]: each object $B^{\otimes n}$ is self-dual, and the dagger of a pure map $f : B^{\otimes n} \to B^{\otimes k}$ is its adjoint $f^\dagger : B^{\otimes k} \to B^{\otimes n}$.

Operationally, we are interested in representing circuits built from the following *logical* components, shown here in diagrammatic form (read from bottom to top), next to their interpretation as maps in $\mathbf{LFM}$.

1. The *beam splitter* with parameters $r, t \in \mathbb{C}$, such that $|r|^2 + |t|^2 = 1$:

$$|00\rangle \mapsto |00\rangle, \qquad\qquad |10\rangle \mapsto r|10\rangle + t|01\rangle,$$
$$|01\rangle \mapsto -\bar{t}|10\rangle + \bar{r}|01\rangle, \quad |11\rangle \mapsto |11\rangle.$$

2. The *phase* gate with parameter $\vartheta \in [0, 2\pi)$:

$$|0\rangle \mapsto |0\rangle, \quad |1\rangle \mapsto e^{i\vartheta}|1\rangle.$$
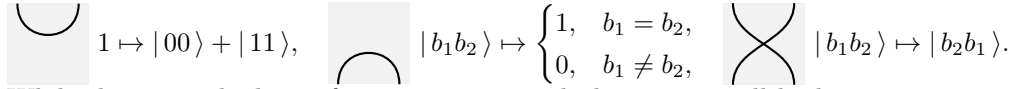
3. The *fermionic swap* gate:

$$|00\rangle \mapsto |00\rangle, \quad |10\rangle \mapsto |01\rangle,$$
$$|01\rangle \mapsto |10\rangle, \quad |11\rangle \mapsto -|11\rangle.$$

4. *Empty state* and *occupied state* preparation:

 $\quad 1 \mapsto |\,0\,\rangle, \quad\quad\quad$ $\quad 1 \mapsto |\,1\,\rangle.$

All of these are isometries, which makes them, at least in principle, physically implementable gates; see for example [19] for the description of an electron beam splitter.

Apart from the fermionic swap gate, which exploits the antisymmetry of fermionic particles under exchange, these operations are structurally the same as those used in implementations of linear optical quantum computing (LOQC), such as the Knill-Laflamme-Milburn scheme [25], which employ photons, that is, bosonic particles as resources. The two models seem closely related; given the way that the fermionic swap ties the other components together in our axiomatisation, and that the impossibility for two particles to occupy the same mode – a constraint for the bosons in LOQC – is simply a consequence of Pauli exclusion for fermions, it seems possible to us that the logical features of the optical model are a consequence of the features of the fermionic model.

In addition to the logical components, we need the following structural components – the *dualities* and the *swap* – which allow us to treat all our diagrams as components of a circuit diagram, which can be connected together in an undirected fashion, permuting and transposing their inputs and outputs:

 $1 \mapsto |\,00\,\rangle + |\,11\,\rangle, \quad$ $|\,b_1 b_2\,\rangle \mapsto \begin{cases} 1, & b_1 = b_2, \\ 0, & b_1 \neq b_2, \end{cases} \quad$ $|\,b_1 b_2\,\rangle \mapsto |\,b_2 b_1\,\rangle.$

While the swap, dualities, fermionic swap, and phase gates will be basic components of our diagrammatic calculus, we are going to further decompose beam splitters and state preparations. The components so obtained may not correspond to physical operations by themselves, but they have the property that the result of transposing or swapping any of their inputs or outputs only depends on the final number of inputs and outputs. This allows us to treat their diagrammatic representations as vertices of an undirected vertex-labelled multigraph: only the overall arity matters. In addition to making calculations simpler, this enables one to implement the calculus in graph rewriting software, such as Quantomatic [24].

The additional components, given here in "all-output form" together with their interpretation in **LFM**, are the following.

1. The binary and ternary black vertex:

 $\quad 1 \mapsto |\,10\,\rangle + |\,01\,\rangle, \quad\quad$ $\quad 1 \mapsto |\,100\,\rangle + |\,010\,\rangle + |\,001\,\rangle.$

2. The binary white vertex with parameter $z \in \mathbb{C}$:

 $\quad 1 \mapsto |\,00\,\rangle + z|\,11\,\rangle.$
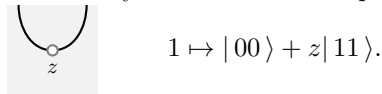
▶ **Remark 4.** Up to a normalising factor, the interpretations of the binary and ternary vertex are known as EPR state and W state, respectively, in qubit theory [12].

When we draw black and white vertices with a different partition of inputs and outputs, we assume that a particular partial transposition has been fixed, for example to the left:

 $:=$ $,\quad\quad$ $:=$ $.$
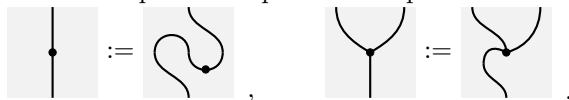
Now, a phase gate with parameter ϑ is simply a binary white vertex with parameter $e^{i\vartheta}$. The beam splitter with parameters r, t, and the state preparations can be decomposed as follows:

We also introduce a simplified notation for a composite diagram that plays an important role in our calculus, whose interpretation is the *projector* on the even subspace of two LFMs:

$$|00\rangle \mapsto |00\rangle, \qquad |11\rangle \mapsto |11\rangle,$$
$$|01\rangle, |10\rangle \mapsto 0.$$
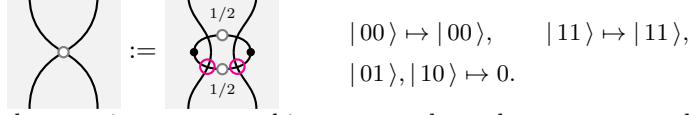
As the notation suggests, this corresponds to the quaternary white vertex of the original ZW calculus. Similarly to the black and white vertices already introduced, its interpretation is symmetric under transposition and swapping of inputs and outputs, so we can freely draw quaternary white vertices with a different partition of inputs and outputs.

▶ **Remark 5.** Our calculus does not include measurements, probabilistic mixing, or any kind of classical control as internal operations. In future work, we hope to extend our axiomatisation to a mixed quantum-classical calculus, in the style of [8] (see [7, Chapter 8] for a more recent version), incorporating all these elements.

For now, we can calculate the probability of detecting particles at the output ends of a circuit by closing the circuit with occupied and empty state diagrams; a closed circuit is then interpreted as a map $\mathbb{C} \to \mathbb{C}$, that is, a scalar. This will be the probability amplitude of detecting a particle where we have closed with an occupied state, and not detecting it where we have closed with an empty state.

To reason rigorously about our diagrammatic calculus, we rely on the theory of PROs (PROduct categories) [26], strict monoidal categories that have $\mathbb{N}$ as set of objects, and monoidal product given, on objects, by the sum of natural numbers. Morphisms $n \to m$ in a PRO represent operations with n inputs and m outputs. Given a *monoidal signature*, that is, a set of operations with arities $T := \{f_i : n_i \to m_i\}_{i \in I}$, one can generate the free PRO $F[T]$ on T, whose operations are free sequential and parallel compositions of the f_i, modulo the axioms of monoidal categories. By a classic result of Street and Joyal [20, Theorem 1.2], this is equivalent to the PRO whose morphisms are obtained by horizontally juxtaposing and vertically plugging string diagrams with the correct arity, one for each generator, then quotienting by planar isotopy of diagrams. Thus, in the remainder, we will not distinguish between the two, identifying diagrams and operations.

▶ **Definition 6.** Let T be the monoidal signature with operations $\mathsf{swap} : 2 \to 2$, $\mathsf{dual} : 0 \to 2$, $\mathsf{dual}^\dagger : 2 \to 0$, $\mathsf{fswap} : 2 \to 2$, $\mathsf{black}_2 : 0 \to 2$, $\mathsf{black}_3 : 0 \to 3$, and $\mathsf{white}_z : 0 \to 2$, for all $z \in \mathbb{C}$. The *language* of the fermionic ZW calculus is the free PRO $F[T]$.

The correspondence with the diagrammatic components we listed earlier should be self-explanatory, and their interpretation induces a monoidal functor $f : F[T] \to \mathbf{LFM}$.

Given a set E of equations between diagrams with the same arity in $F[T]$, we can quotient $F[T]$ by the smallest equivalence relation including E and compatible with composition and monoidal product, to obtain a PRO $F[T/E]$, together with a quotient functor $p_E : F[T] \to F[T/E]$.

▶ **Definition 7.** The interpretation $f : F[T] \to \mathbf{LFM}$ is *universal* if it is a full functor. A set E of equations is *sound* for $f : F[T] \to \mathbf{LFM}$ if f factors as $f_E \circ p_E$ for a functor $f_E : F[T/E] \to \mathbf{LFM}$. A sound set of equations is *complete* for $\mathbf{LFM}$ if f_E is an equivalence of monoidal categories.

In the next section, we introduce the axioms of the fermionic ZW calculus, in the form of equations between diagrams of $F[T]$; it can be checked that they are all sound for the interpretation. We will later show that they are also complete. This means that whenever two diagrams of the fermionic ZW calculus are "extensionally equal", that is, they have the same interpretation in **LFM**, one can be rewritten into the other by applying a finite sequence of equations.

3   Axioms and derived equations

We divide the set E of axioms into four groups, based on the generators to which they mainly pertain.

▶ **Axioms 8.** *Structural axioms.*
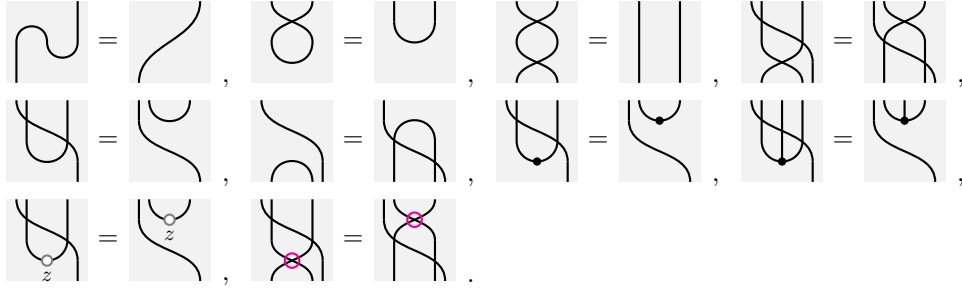
Together, these axioms imply that the swap and dualities make $F[T/E]$ a compact closed category on a self-dual object. The Kelly-Laplaza coherence theorem [22, Theorem 8.2] then allows us to extend our topological reasoning to the swapping and transposition of wires.

▶ **Axioms 9.** *Axioms for the fermionic swap.*
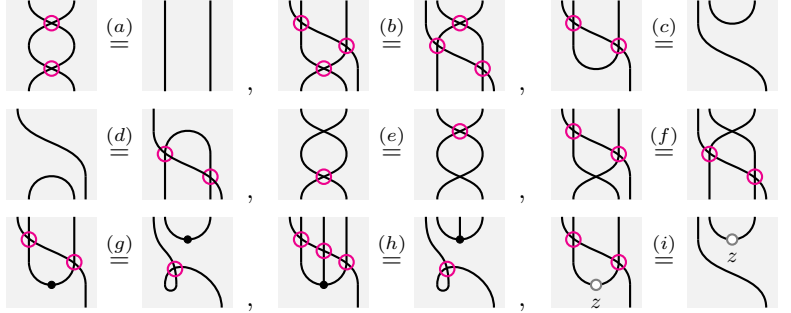
These axioms say that the fermionic swap behaves like a symmetric braiding in $F[T/E]$, except for the fact that sliding the black vertices (that is, the only odd generators) through a wire induces a fermionic "self-crossing" on it.

Moreover, the axioms on the interplay between the structural and fermionic swaps imply that only the number of fermionic swaps between two wires matters, and not their direction; which, as we will see, also implies that a sequence of two fermionic self-crossings on either side of a wire can be straightened.

Altogether, the result of the other axioms is that any diagram containing an *even* number of black vertices can slide past a wire through fermionic swaps with no other effect, while any diagram containing an *odd* number of black vertices can do the same by introducing a fermionic self-crossing on the wire. As with the structural axioms, we will make use of this fact implicitly most of the time.

▶ **Axioms 10.** *Axioms for black vertices.*
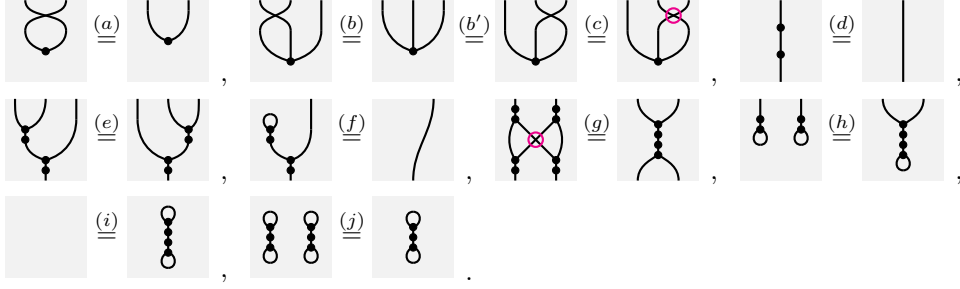
These axioms say that the black vertices are symmetric under permutation of wires (which justifies, a posteriori, their arbitrary transposition), and that they can be assembled to form a (co)commutative (co)monoid. This (co)monoid has the property of forming a bialgebra (in fact, a Hopf algebra) with its own transpose.

In the interpretation, this is the Hopf algebra known as *fermionic line* in the theory of quantum groups [27, Example 14.6], whose comultiplication is given by $|0\rangle \mapsto |00\rangle$, $|1\rangle \mapsto |10\rangle + |01\rangle$. As discussed in [17, Section 5.3], the fermionic line has "anyonic" and "bosonic" analogues in every countable dimension, with the same self-duality property.

The final axiom says that 0 times 0 is 0; it will serve to ensure that there is a unique zero map between any two systems, rather than an "even" and an "odd" zero map.

▶ **Axioms 11.** *Axioms for white vertices.*
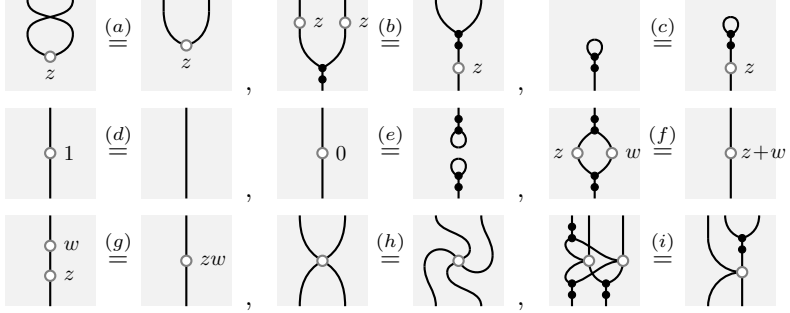
These axioms say that the binary white vertices are endomorphisms for the fermionic line algebra, and that composition and convolution by the algebra correspond to product and sum, respectively, of their complex parameters. Finally, the projector is symmetric under cyclic permutation of its wires, and it determines a kind of mixed action/coaction of the algebra on itself.

▶ **Remark 12.** Because **LFM** is a subcategory of the category **Qubit** of [18], all the axioms of the fermionic ZW calculus are sound for the original ZW calculus. Moreover, adding either the ternary or the unary white vertex from the original ZW calculus to our language would make it universal for **Qubit**. We have not yet investigated, however, what axioms would need to be added to E in the extended signature to make it complete.

We state some useful derived equations, leaving the proofs to the Appendix.

▶ **Proposition 13.** *The following equations hold in $F[T/E]$:*
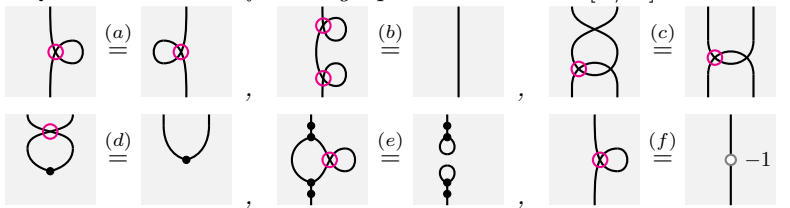
▶ **Proposition 14.** *The following equations hold in* $F[T/E]$:
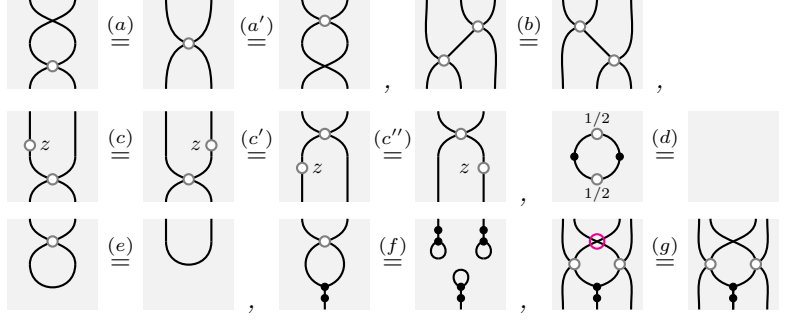
Together with its invariance under cyclic permutation of wires, the first two equations justify the arbitrary transposition of inputs and outputs of the quaternary white vertex.

Our axioms form a sound and complete set of equations for **LFM**, so in principle any equation of diagrams whose interpretations are equal can be derived from them. In practice, however, it is convenient to introduce further short-hand notation, including black vertices with n wires and white vertices with $2n$ wires for all $n \in \mathbb{N}$, and derive inductive equation schemes to use directly in proofs.

1. *Black vertices.* The nullary and unary black vertices are defined as follows:

We already have binary and ternary black vertices. For $n > 3$, the n-ary black vertex is defined inductively, together with its interpretation in **LFM**, by

$$\text{(diagram)} := \text{(diagram)} \qquad 1 \mapsto \sum_{k=1}^{n} |\underbrace{0\ldots0}_{k-1}1\underbrace{0\ldots0}_{n-k}\rangle.$$

Here, and in what follows, lighter wires and vertices indicate the repetition of a pattern for a number of times, which may or may not be specified. This is similar to the the way that "..." is often used, and can be formalised using !-boxes in pattern graphs, as developed in [23].

2. *White vertices.* The nullary white vertex with parameter $z \in \mathbb{C}$ is defined by

We already have binary white vertices with parameters. For $n > 1$, the $2n$-ary white vertex with parameter $z \in \mathbb{C}$ is defined inductively, together with its interpretation in **LFM**, by

$$\text{(diagram)} := \text{(diagram)} \qquad 1 \mapsto |\underbrace{0\ldots0}_{2n}\rangle + z|\underbrace{1\ldots1}_{2n}\rangle.$$

We state some basic properties of black and white vertices. Both are symmetric under permutation of wires, which allows us to write vertices with different numbers of inputs and outputs, transposing some of them with no ambiguity. Most importantly, they satisfy certain "fusion" equations, as shown on the first two lines. All black vertices correspond to odd maps, while white vertices correspond to even maps, as reflected in their sliding through fermionic swaps, on the fourth line; finally, black vertices are unaffected by fermionic swaps of their wires, whereas the sign of the white vertex parameter is flipped.

▶ **Proposition 15.** *The following equations hold in* $F[T/E]$ *for black and white vertices of any arity:*
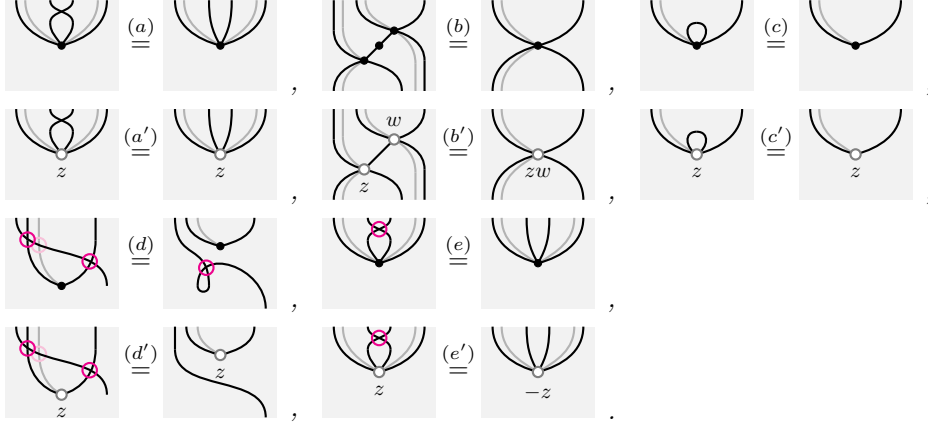
Several other equations, both axioms and derived, admit inductive generalisations; we list them in the following Proposition.

▶ **Proposition 16.** *The following equations hold in $F[T/E]$ for black and white vertices of any compatibile arities:*
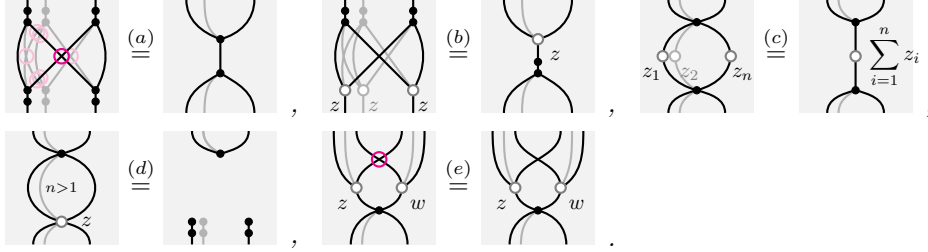
▶ **Remark 17.** Some of these inductive schemes subsume several axioms at once: for example, Proposition 16.(a) has Axioms 10.(g), 10.(h), and 10.(i) as special cases, and Proposition 16.(b) has Axioms 11.(b), 11.(c), and 11.(i) as special cases.

4 Normal form and completeness

We prove completeness in three stages:

1. First, we associate to any state $v : \mathbb{C} \to B^{\otimes n}$ of **LFM** a diagram $g(v) : 0 \to n$ in $F[T]$ such that $f(g(v)) = v$. Because both categories are compact closed, and the dualities of **LFM** are in the image of f, this assignment can be extended to any map of **LFM**, proving universality of our interpretation. We say that a diagram is in *normal form* if it is of the form $g(v)$ for some v.

2. Then, we show that any composite of diagrams in normal form can be rewritten in normal form using the equations in E, proving that g determines a monoidal functor from **LFM** to $F[T/E]$.

3. Finally, we show that all the generators of $F[T]$ can be rewritten in normal form using the equations in E, proving that g and $f_E : F[T/E] \to \mathbf{LFM}$ are two sides of a monoidal equivalence between $F[T/E]$ and **LFM**.

▶ **Theorem 18** (Universality). *The functor $f : F[T] \to \mathbf{LFM}$ is full.*

Proof. Write an arbitrary state $v : \mathbb{C} \to B^{\otimes n}$ in the form $1 \mapsto \sum_{i=1}^{m} z_i | b_{i1} \dots b_{in} \rangle$, where $z_i \neq 0$ for all i, and no pair of n-tuples $(b_{i1}, \dots, b_{in})$ is equal; we can fix an ordering (for

example, lexicographic) on n-tuples of bits to make this unique. Then, define

$$g(v) := \quad \text{if } v \text{ is odd,} \qquad \text{if } v \text{ is even,} \qquad (1)$$

where, for $i = 1, \ldots, m$ and $j = 1, \ldots, n$, the dotted wire connecting the i-th white vertex to the j-th output is present if and only if $b_{ij} = 1$. The definition is only ambiguous if $v = 0$, in which case we arbitrarily pick one of the two forms; they will be equal in $F[T/E]$ by Axiom 10.(j).

Because for all summands of an odd (respectively, even) state v, we have $b_{ij} = 1$ for an odd (respectively, even) number of bits, the white vertices in $g(v)$ have an odd (respectively, even) number of outputs. The two distinct forms of $g(v)$ for odd and even states ensure that only white vertices with an even arity appear.

It can then be checked that $f(g(v)) = v$, which, by our earlier remark, suffices to prove the statement. ◀

▶ **Definition 19.** A string diagram of $F[T]$ is in *normal form* if it is $g(v)$ for some state v of **LFM**. It is in *pre-normal form* if it has one of the two forms in (1), where the following are also allowed:

- the white vertices can be in an arbitrary order;
- two or more white vertices may be connected to the exact same outputs;
- z_i may be 0 for some i.

The completeness proof closely follows that of the qubit ZW calculus [17, Section 5.2]. The one proof that is significantly different is the following. We take the liberty of "zooming in" on a certain portion of a diagram, which may require some reshuffling of vertices, using swapping or transposition of wires, with the implicit understanding that this can always be reversed later.

▶ **Lemma 20** (Negation). *The composition of one output of a diagram in pre-normal form with a binary black vertex can be rewritten in pre-normal form, and that has the effect of "complementing" the connections of the output to white vertices: that is, locally,*

.
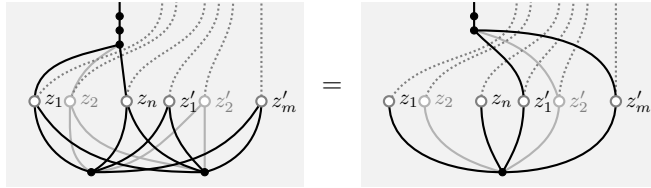
▶ Remark. In the picture, the dotted wires can stand for a multitude of wires. The version where the original diagram is odd, rather than even, is obtained by composing again both sides with a binary black vertex and using Axiom 10.(d).

Proof. Using the "fusion equations" Proposition 15.(b) and (b'), we rewrite the left-hand side as

.
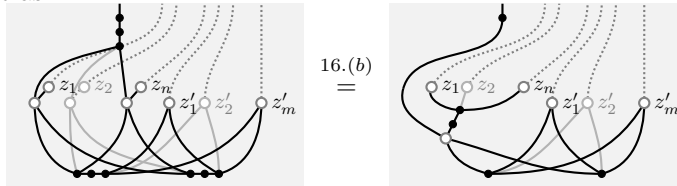
By definition of the quaternary white vertex, this is equal to
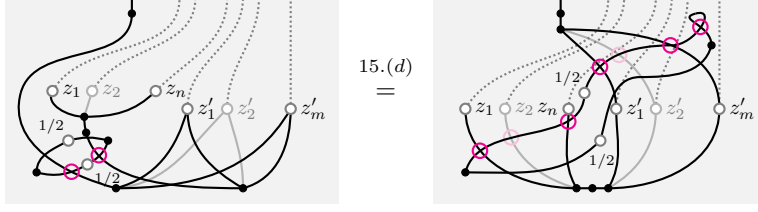
where we made implicit use of some symmetry properties of vertices. Now, fusing black vertices, and using Proposition 15.(d) and (d') to move the closed loop to the outside of the main diagram, we see that this is equal to
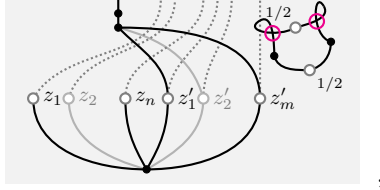
and we can conclude by Proposition 13.(b) and 14.(d). ◀

In the following, and later statements, "plugging one output of a diagram into another" means a post-composition with $\mathsf{dual}^\dagger : 2 \to 0$, possibly after some swapping of wires.

▶ **Lemma 21** (Trace). *The plugging of two outputs of a diagram in pre-normal form into each other can be rewritten in pre-normal form.*

Proof. Essentially the same as [17, Lemma 5.24]. ◀

The nullary black vertex is interpreted as the scalar 0; the following lemma shows that it acts as an "absorbing element" for diagrams in pre-normal form.

▶ **Lemma 22** (Absorption). *For all diagrams in pre-normal form, the following equation holds in $F[T/E]$:*

 (2)
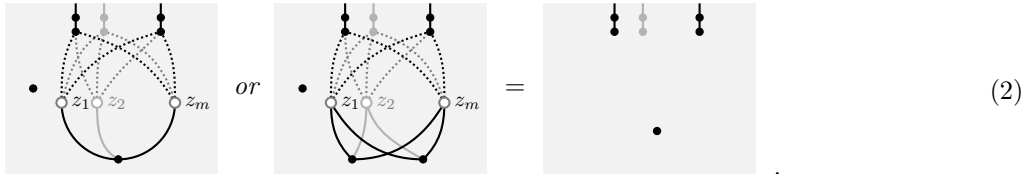
Proof. If the diagram is even, expanding the nullary black vertex, we can treat it as an additional output of the diagram, with no connections to the white vertices, composed with a unary black vertex; then the proof is the same as [17, Lemma 5.25].

Suppose the diagram is odd. If it has at least one output wire, we can freely introduce two binary black vertices on it; applying the negation lemma once, we obtain a negated even diagram, to which the first part of the proof can be applied. Another application of the negation lemma, followed by Axiom 10.(j), produces the desired equation. If the diagram has no outputs, it necessarily consists of a single nullary black vertex, and the statement follows immediately from Axiom 10.(j). ◀

▶ **Lemma 23** (Functoriality of the normal form). *Any composition of two diagrams in pre-normal form can be rewritten in pre-normal form.*

Proof. We can factorise any composition of diagrams in pre-normal form as a tensor product followed by a sequence of "self-pluggings"; thus, by the trace lemma, it suffices to prove that a tensor product – diagrammatically, the juxtaposition of two diagrams in pre-normal form – can be rewritten in pre-normal form.

Suppose first that the two diagrams are both even. We can create a pair of unary black vertices connected by a wire by Axiom 10.(i), and treat them as additional outputs, one for each diagram. In that case, the proof proceeds exactly as [17, Theorem 5.26].

Now, suppose one diagram is odd, or they both are odd. If the odd diagrams have at least one output wire, we can introduce a pair of black vertices on it, and apply the negation lemma to produce negated even diagrams. We can then apply the first part of the proof to obtain a diagram in pre-normal form negated once or twice, then apply the negation lemma again to conclude. If one of the odd diagrams has no outputs, it necessarily consists of a single nullary black vertex, and we can conclude with an application of the absorption lemma. ◀

▶ **Lemma 24.** *Any diagram in pre-normal form can be rewritten in normal form.*

Proof. If the diagram is odd, the proof of [17, Lemma 5.22] goes through. If the diagram is even, and has at least one output wire, we can introduce a pair of binary black vertices, apply the negation lemma once to produce a negated odd diagram, reduce that to normal form, and apply the negation lemma again; it is easy to see that negation turns diagrams in normal form into diagrams in normal form, *modulo* a reshuffling of white vertices.

If the diagram has no output wires, then it is of the form

where the right-hand side is in normal form. This concludes the proof. ◀

▶ **Theorem 25** (Completeness). *The functor $f_E : F[T/E] \to$ **LFM** induced by the soundness of E for the interpretation $f : F[T] \to$ **LFM** is a monoidal equivalence.*

Proof. By the combination of the previous two lemmas, it suffices to show that all the generators (with all wires transposed to output wires) can be rewritten in pre-normal form. For the ternary and binary black vertices,

For the binary white vertex with parameter $z \in \mathbb{C}$,

By Axiom 11.(d), the rewriting of dualities in normal form follows as a special case of the binary white vertex with parameter 1.

For the fermionic swap, we use the fact that we know how to rewrite the tensor product of two dualities in normal form:

The case of the structural swap is similar, and easier. This concludes the proof. ◀

▶ **Remark 26.** We can make this an equivalence of dagger compact closed categories, by defining the dagger of a morphism in $F[T/E]$, represented by a diagram in $F[T]$, to be the vertical reflection of that diagram, with parameters $z \in \mathbb{C}$ of white vertices turned into their complex conjugates $\overline{z}$. For example,

$$\text{(diagram)} \quad \overset{(-)^\dagger}{\mapsto} \quad \text{(diagram)} \ .$$
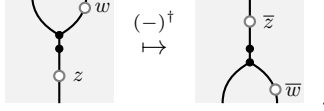
▶ **Remark 27.** The only properties of complex numbers that are used in the proof are that they form a commutative ring, and that they contain an element z such that $z + z = 1$ (namely, $1/2$). Thus, we can replace $\mathbb{C}$ with any commutative ring R that has the latter property (for example, $\mathbb{Z}_{2n+1}$, for each $n \in \mathbb{N}$), and obtain a similar completeness result for "LFMs with coefficients in R".

Moreover, for any such ring, instead of introducing binary white vertices with arbitrary parameters $r \in R$, we can introduce one binary white vertex for each element of a family of generators of R, together with one axiom for each relation that they satisfy. Then, in the normal form, instead of having a white vertex labelled $r \in R$ at each end of the bottom black vertex or vertices, we will need to have some expression of r by sums and products of generators, encoded by composition and convolution by the fermionic line algebra.

The completeness proof still goes through: we work with diagrams in pre-normal form, where terms in a sum of products of generators are decomposed into different legs of the bottom vertex or vertices, until the very end; then Lemma 24 can be adapted to combine white vertices with the same connections into a fixed expression of the sum of their parameters.

For example, in the complex case, it may be convenient to have separate phase gates, that is, white vertices with parameter $e^{i\vartheta}$, for $\vartheta \in [0, 2\pi)$, and "resistor" gates, with real parameter $r > 0$.

▶ **Remark 28.** It is customary to describe the fermionic behaviour of a multi-particle system in terms of a pair of operators $a^\dagger$ (creation) and a (annihilation) that satisfy the anti-commutation relation $aa^\dagger = 1 - a^\dagger a$; see for example [32, Chapter 27]. In our language, these operators can be defined as
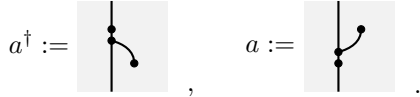
$$a^\dagger := \text{(diagram)} \ , \qquad a := \text{(diagram)} \ .$$

We can see the anti-commutation relation as subsumed by the axioms in the following way: pulling back the linear structure of **LFM** to $F[T/E]$ through the equivalence, we have

$$\text{(diagram)} \ = \ \text{(diagram)} \ - \ \text{(diagram)} \ , \tag{3}$$

from which we obtain

$$\text{(diagram)} \ \overset{10.(g)}{=} \ \text{(diagram)} \ = \ \text{(diagram)} \ \overset{(3)}{=} \ \text{(diagram)} \ - \ \text{(diagram)} \ ,$$

which can be read as the equation $aa^\dagger = 1 - a^\dagger a$.

5 An application: the Mach-Zehnder interferometer

The Mach-Zehnder interferometer is a classic quantum optical setup (see for example [30, Chapter 4]), which, despite its simplicity, can demonstrate interesting features of quantum mechanics, as in the Elitzur-Vaidman bomb tester experiment [14]. The theoretical setup can

be straightforwardly imported into FQC, with the same statistics as long as single-particle experiments are concerned; an electronic analogue of the Mach-Zehnder interferometer has also been realised in practice [19].
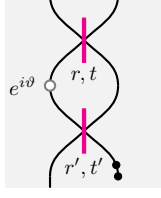
With the graphical notation introduced in Section 2, the experimental setup is represented by the diagram on the left, where $r, t, r', t' \in \mathbb{C}$ and $\vartheta \in [0, 2\pi)$ are parameters subject to $|r|^2 + |t|^2 = |r'|^2 + |t'|^2 = 1$. In practice, it would also include "mirrors", or beam splitters with $|r| = 1$, which we omit in the picture, instead taking the liberty of bending wires at will.

As a first application of the fermionic ZW calculus, we show how this circuit diagram can be simplified in just a few steps using our axioms, in such a way that its statistics become immediately readable from the diagram.

In our language, the diagram becomes
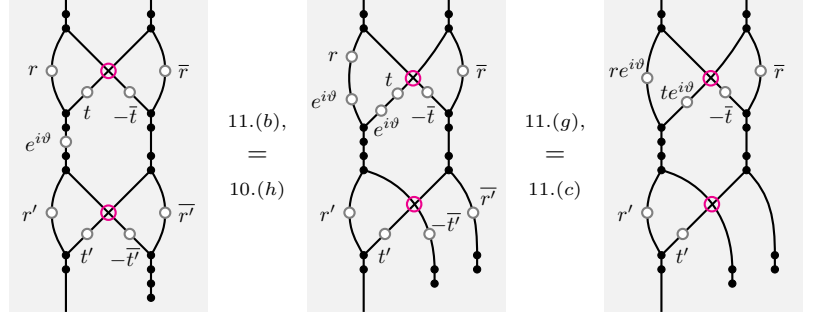
which, sliding the leftmost empty state past the fermionic swap, and using Axiom 10.(f) twice, becomes
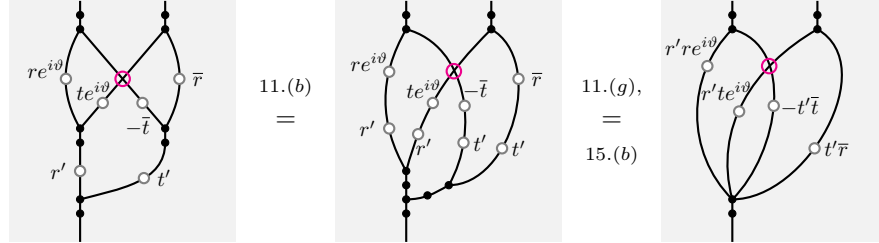
Finally, using the fermionic swap symmetry of black vertices (Proposition 15.(e)), together with Proposition 16.(c), this simplifies to
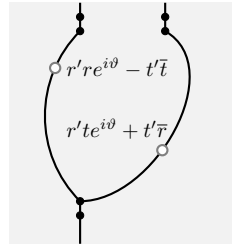
If we input one particle, after fusing the bottom black vertices, we obtain a diagram in normal form, whose interpretation in **LFM** we can read off as $1 \mapsto (r're^{i\vartheta} - t'\overline{t})\,|\,10\,\rangle + (r'te^{i\vartheta} + t'\overline{r})\,|\,01\,\rangle$.

So, the probability of detecting the particle at the left-hand output is $|r're^{i\vartheta} - t'\overline{t}|^2$, and the probability of detecting the particle at the right-hand output is $|r'te^{i\vartheta} + t'\overline{r}|^2$. If the beam splitters are symmetric, that is, $r = r' = \frac{1}{\sqrt{2}}$, and $t = t' = \frac{i}{\sqrt{2}}$, the probability amplitudes

become

$$\frac{1}{2}(e^{i\vartheta} - 1) = e^{i(\frac{\vartheta+\pi}{2})} \sin\vartheta, \qquad \frac{i}{2}(e^{i\vartheta} + 1) = e^{i(\frac{\vartheta+\pi}{2})} \cos\vartheta,$$

leading to probabilities $\sin^2\vartheta$ of detecting the particle at the left-hand output, and $\cos^2\vartheta$ of detecting it at the right-hand output.

Arguably, given that this particular example involves at most binary gates, a matrix calculation would not have been considerably harder. On the other hand, the result appears here as the outcome of a short sequence of intuitive, algebraically motivated local steps, rather than the unexplained product of a large matrix multiplication. We expect the advantage to become clearer when implementations of rewrite strategies in graph rewriting software are used to simplify larger circuits.

6 Conclusions and outlook

In this paper, we introduced a string-diagrammatic language for circuits of local fermionic modes, together with equations that axiomatise their theory of extensional equality: that is, two diagrams represent the same linear map of local fermionic modes if and only if they are equal modulo the equations. We believe that these fermionic circuits are to the ZW calculus what Clifford circuits [1] are to the ZX calculus: not the largest family of circuits that can technically be represented, but the one whose basic gates have simple, natural representations in terms of the language's components.

There are still several open questions and directions on the "syntactic" side. We do not know whether all our axioms are independent, nor we have looked at rewrite strategies, or ways of orienting the equations, beyond the goal of proving completeness. There is, then, the question of variants and extensions: we have mentioned a potential extension to mixed-state processes, via a mixed quantum-classical calculus in the style of [7, Chapter 8]; moreover, both universality and completeness are open problems for anyonic and bosonic generalisations of the fermionic ZW calculus, in the style of [17, Section 5.3].

The greatest challenge, however, is finding "real-world" applications for the calculus. With the Mach-Zehnder interferometer, we have only given a toy example, perhaps useful for pedagogical purposes, but we have not even attempted to link our work to current research on algorithms or complexity in FQC. The first version of a ZW calculus was introduced in order to tackle open problems in the classification of multipartite entanglement [6]: as a first step, the fermionic ZW calculus, with its strong topological flavour, involving braidings and a single type of ternary vertices, may be a better testing ground for this approach.

References

1   M. Backens. The ZX-calculus is complete for stabilizer quantum mechanics. *New Journal of Physics*, 16(9):093021, 2014. `doi:10.1088/1367-2630/16/9/093021`.

2   M.-C. Bañuls, J.I. Cirac, and M.M. Wolf. Entanglement in fermionic systems. *Physical Review A*, 76(2), 2007. `doi:10.1103/physreva.76.022311`.

3   S.B. Bravyi and A. Yu. Kitaev. Fermionic quantum computation. *Annals of Physics*, 298(1):210–226, 2002. `doi:10.1006/aphy.2002.6254`.

4   B. Coecke and R. Duncan. Interacting quantum observables. In *Automata, Languages and Programming*, pages 298–310. Springer, 2008. `doi:10.1007/978-3-540-70583-3\_25`.

5   B. Coecke, R. Duncan, A. Kissinger, and Q. Wang. Strong complementarity and non-locality in categorical quantum mechanics. In *2012 27th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 2012. `doi:10.1109/lics.2012.35`.

6   B. Coecke and A. Kissinger. The compositional structure of multipartite quantum entanglement. In *Automata, Languages and Programming*, pages 297–308. Springer Berlin Heidelberg, 2010. `doi:10.1007/978-3-642-14162-1_25`.

7   B. Coecke and A. Kissinger. *Picturing Quantum Processes*. Cambridge University Press (CUP), 2017. `doi:10.1017/9781316219317`.

8   B. Coecke and D. Pavlovic. Quantum measurements without sums. In *Chapman & Hall/CRC Applied Mathematics & Nonlinear Science*, pages 559–596. Chapman and Hall/CRC, 2007. `doi:10.1201/9781584889007.ch16`.

9   B. Coecke, D. Pavlovic, and J. Vicary. A new description of orthogonal bases. *Mathematical Structures in Computer Science*, 23(03):555–567, 2012. `doi:10.1017/s0960129512000047`.

10 G.M. D'Ariano, F. Manessi, P. Perinotti, and A. Tosini. The Feynman problem and fermionic entanglement: Fermionic theory versus qubit theory. *International Journal of Modern Physics A*, 29(17):1430025, 2014. `doi:10.1142/s0217751x14300257`.

11 R. Duncan and S. Perdrix. Rewriting measurement-based quantum computations with generalised flow. In *Automata, Languages and Programming*, pages 285–296. Springer Berlin Heidelberg, 2010. `doi:10.1007/978-3-642-14162-1_24`.

12 W. Dür, G. Vidal, and J.I. Cirac. Three qubits can be entangled in two inequivalent ways. *Physical Review A*, 62(6), 2000. `doi:10.1103/physreva.62.062314`.

13 V. Eisler and Z. Zimborás. On the partial transpose of fermionic gaussian states. *New Journal of Physics*, 17(5):053048, 2015. `doi:10.1088/1367-2630/17/5/053048`.

14 A.C. Elitzur and L. Vaidman. Quantum mechanical interaction-free measurements. *Foundations of Physics*, 23(7):987–997, 1993. `doi:10.1007/bf00736012`.

15 N. Friis, A.R. Lee, and D.E. Bruschi. Fermionic-mode entanglement in quantum information. *Physical Review A*, 87(2), 2013. `doi:10.1103/physreva.87.022338`.

16 A. Hadzihasanovic. A diagrammatic axiomatisation for qubit entanglement. In *Proceedings of the 30th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '15, pages 573–584. IEEE, 2015. `doi:10.1109/LICS.2015.59`.

17 A. Hadzihasanovic. *The algebra of entanglement and the geometry of composition*. PhD thesis, University of Oxford, 2017. Available at `https://arxiv.org/abs/1709.08086`.

18 A. Hadzihasanovic, K.F. Ng, and Q. Wang. Two complete axiomatisations of pure-state qubit quantum computing, 2018. Accepted at the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) 2018.

19 Y. Ji, Y. Chung, D. Sprinzak, M. Heiblum, D. Mahalu, and H. Shtrikman. An electronic Mach–Zehnder interferometer. *Nature*, 422(6930):415–418, 2003. `doi:10.1038/nature01503`.

20 A. Joyal and R. Street. The geometry of tensor calculus, I. *Advances in Mathematics*, 88(1):55–112, 1991. `doi:10.1016/0001-8708(91)90003-p`.

21 L.H. Kauffman. *Knots and Physics*. World Scientific Publishing Co. Pte. Ltd., 2001. `doi:10.1142/9789812384836`.

22 G.M. Kelly and M.L. Laplaza. Coherence for compact closed categories. *Journal of Pure and Applied Algebra*, 19:193–213, 1980. `doi:10.1016/0022-4049(80)90101-2`.

23 A. Kissinger, A. Merry, and M. Soloviev. Pattern graph rewrite systems. *Electronic Proceedings in Theoretical Computer Science*, 143:54–66, 2014. `doi:10.4204/eptcs.143.5`.

24 A. Kissinger and V. Zamdzhiev. Quantomatic: A proof assistant for diagrammatic reasoning. In *Automated Deduction - CADE-25*, pages 326–336. Springer International Publishing, 2015. `doi:10.1007/978-3-319-21401-6_22`.

25 E. Knill, R. Laflamme, and G.J. Milburn. A scheme for efficient quantum computation with linear optics. *Nature*, 409(6816):46–52, 2001.

26 S. Lack. Composing PROPs. *Theory and Applications of Categories*, 13(9):147–163, 2004.

27 S. Majid. *A Quantum Groups Primer*. Cambridge University Press, 2002. `doi:10.1017/cbo9780511549892`.

28 K.F. Ng and Q. Wang. A universal completion of the ZX-calculus. *arXiv preprint arXiv:1706.09877*, 2017.

29 M.A. Nielsen and I.L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2009. `doi:10.1017/cbo9780511976667`.

30 M.O. Scully and M.S. Zubairy. *Quantum optics*. Cambridge University Press, 1997. `doi:10.1017/cbo9780511813993`.

31 P. Selinger. Finite dimensional Hilbert spaces are complete for dagger compact closed categories. *Electronic Notes in Theoretical Computer Science*, 270(1):113–119, 2011. `doi:10.1016/j.entcs.2011.01.010`.

32 P. Woit. *Quantum Theory, Groups and Representations*. Springer International Publishing, 2017. `doi:10.1007/978-3-319-64612-1`.

A   Proofs of derived equations

Proposition 13. Equation (a) comes from the following manipulation:

Equation (b) then follows from (a), combined with the equation

which is a consequence of the fermionic swap axioms by the Whitney trick [21, p. 484]. Equation (c) is proved by the following argument:

whereas (d) comes from

finally using the symmetry of the black vertex under the structural swap.

Equation (e) is proved by the following argument:

where we tacitly used Axiom $10.(d)$ to introduce or eliminate pairs of binary black vertices in several occasions.

Finally, for equation (f), start by considering that

by Axioms $10.(e)$ and $11.(d)$, this is equal to

This completes the proof.   ◀

Proof of Proposition 14. Substituting the definition of the projector, Axiom 11.(h) becomes the following equation:

$$\tag{4}$$

Equations (a) and (a') are then immediate consequences of Proposition 13.(c) and its transposes, applied to the right-hand side of (4).

Equation (b) is also immediate from the definition: because swaps slide through fermionic swaps and vice versa, we can slide one "circle" past another to get

For equations (c), (c'), and (c''), we use either of the forms in equation (4) and slide the binary white vertex through a fermionic swap using Axiom 9.(i), to move it to a different wire.

Equation (d) comes from

finally applying Axiom 10.(i). Then, equation (e) follows from it by

In order to prove equation (f), consider first that

$$\tag{5}$$

and we can eliminate the circle by equation (d). Then,

Finally, for equation (g), observe that the projector contains an even number of black vertices, hence it can slide past fermionic swaps with no other effect. Therefore,

This concludes the proof.   ◀

Proof of Proposition 15. All the equations are proved by induction on the arity of the vertices involved.

For equation (a), let n be the number of outputs of the black vertex. For $n = 0, 1$ there is nothing to prove, and for $n = 2, 3$ these are Axioms 10.(a), (b), and (b'). For $n > 3$, if the two swapped wires are the rightmost ones, the equation follows immediately from the ternary case; otherwise, use Axiom 10.(e) on the three rightmost wires, and apply the inductive hypothesis.

For equation (a'), let $2n$ be the number of outputs of the white vertex. For $n = 0$ there is nothing to prove, and $n = 1$ is Axiom 11.(a). For $n > 1$, observe that by Proposition 14.(c), (c') and (c''), we can always move the binary vertex with parameter z to a wire which is not swapped. The case $n = 2$ then follows from the combination of Axiom 11.(h) with Proposition 14.(a) and (a'). For $n > 2$, if the swapped wires are among the three rightmost ones, the equation follows from the case $n = 2$; otherwise, use Proposition 14.(b) (with some wires transposed) on the two rightmost quaternary white vertices, and apply the inductive hypothesis.

Equations (a) and (a') justify the unambiguous writing of n-ary vertices with inputs as well as outputs in equations (b) and (b'), and the latter will follow from the all-output case. In equation (b), let $n, m > 0$ be the arities of the leftmost and rightmost vertex, respectively. If $n = 1$, the equation follows from Axiom 10.(f), and if $n = 2$ from Axiom 10.(d). Suppose $n > 2$. Then, if $m = 1$, the equation follows from Axiom 10.(f), and if $m = 2$ from Axiom 10.(d). All other cases are just immediate from the definition. Equation (b') also follows from the definition, together with Proposition 14.(c), (c') and (c'') in order to move the vertex with parameter w to the wire where the vertex with parameter z is, and Axiom 11.(g) to multiply the two.

In equation (c), let $n > 1$ be the arity of the black vertex in the left-hand side. If $n = 2, 3$ the equation is true by definition. If $n > 3$, by equation (a), we can assume the two wires plugged into each other are the two rightmost ones; the equation then follows from Axiom 10.(f). For equation (c'), let $2n > 1$ be the arity of the white vertex in the left-hand side. If $n = 1$, the equation is true by definition, and if $n = 2$ it follows from Proposition 14.(d), together with Proposition 14.(c), (c') and (c'') to move the vertex with parameter z out of the way. For $n > 2$, by equation (a'), we can assume the two wires plugged into each other are the two rightmost ones, and the equation follows from the case $n = 2$.

Equation (d) is a consequence of Axioms 9.(g) and (h), together with Proposition 13.(b) to eliminate pairs of self-crossings. Equation (d') is a consequence of Axiom 9.(i) together with the definition of the quaternary white vertex.

Equation (e) follows from equation (a) by Axiom 10.(c) and Proposition 13.(d). For equation (e'), let $2n > 1$ be the arity of the white vertex. The case $n = 1$ is a consequence of Proposition 13.(f), and $n = 2$ follows from the following argument:
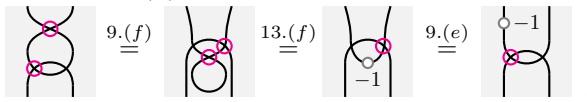
applied to the definition of the quaternary white vertex, as in the right-hand side of (4). All other cases follow from this one, by symmetry.   ◀

Proof of Proposition 16. In equation (a), let n be the number of inputs, and m the number of outputs of the diagrams. The case $n = m = 0$ is Axiom 10.(i), and when either n or $m = 1$, the equation follows from Axiom 10.(d). The cases $n = 0, m > 1$ and $m = 0, n > 1$ are simple inductive generalisations of Axiom 10.(h). Finally, the case $n = m = 2$ is Axiom 10.(g), and from there we can proceed by double induction on n and m, using Proposition 15.(b).

In equation (b), let n be the number of inputs, and $2m - 1$, for $m > 0$, the number of outputs. Suppose first that $m = 1$. The case $n = 0$ is Axiom 11.(c), the case $n = 1$ follows

from Axiom 10.(d), and the case $n = 2$ is Axiom 11.(b); then, for $n > 2$, it is a simple induction starting for the latter. In the case $n = 0$ and $m = 2$,

$$\cdots \overset{11.(c)}{=} \cdots \overset{11.(i)}{=} \cdots \;;$$

by Proposition 15.(b') and (c'), the latter is equal to

$$\cdots \overset{14.(f)}{=} \cdots \overset{10.(i)}{=} \cdots \;.$$

The cases $n = 0$, $m > 2$ are simple inductive generalisations of this one. All cases with $n = 1$ follow from Axiom 10.(d), and the case $n = m = 2$ is Axiom 11.(i). For $n, m > 2$, proceed by double induction, using Proposition 15.(b) and (b').

For equation (c), by Proposition 15.(b) it suffices to prove

$$\cdots \quad = \quad \cdots \sum_{i=1}^{n} z_i \;,$$

which for $n = 0$ is Axiom 11.(e), for $n = 1$ follows from Axiom 10.(d), for $n = 2$ is Axiom 10.(f), and for $n > 2$ is a simple inductive generalisation of the latter.

Similarly, for equation (d), it suffices, by Proposition 15.(b) and (b'), to prove

$$\cdots \quad = \quad \cdots$$

when $m = 0, 1, 2$. If $m = 2$, and $n = 2$, this is Proposition 14.(f), and for $n > 2$ we can proceed by induction, as follows:

$$\cdots \overset{\substack{15.(b'),\\15.(c')}}{=} \cdots \overset{\text{i.h.}}{=} \cdots \overset{\substack{10.(f),\\10.(i)}}{=} \cdots \;.$$

The case $m = 0$, for arbitrary $n > 1$, follows from this one, by

$$\cdots \overset{15.(c')}{=} \cdots \quad = \quad \cdots \overset{10.(i)}{=} \cdots \;,$$

and similarly for the case $m = 1$, where necessarily $n > 2$, by

$$\cdots \quad = \quad \cdots \quad = \quad \cdots \;.$$

Finally, equation (e) is an immediate generalisation of Proposition 14.(g), using Proposition 15.(b) and (b').   ◀

On Repetitive Right Application of B-Terms

Mirai Ikebuchi
Massachusetts Institute of Technology, Cambridge, MA, USA
https://mir-ikbch.github.io/
ikebuchi@mit.edu

Keisuke Nakano[1]
Tohoku University, Sendai, Miyagi, Japan
http://www.riec.tohoku.ac.jp/~ksk/
k.nakano@acm.org

---- **Abstract** ----

B-terms are built from the B combinator alone defined by $B \equiv \lambda f.\lambda g.\lambda x.f\ (g\ x)$, which is well-known as a function composition operator. This paper investigates an interesting property of B-terms, that is, whether repetitive right applications of a B-term cycles or not. We discuss conditions for B-terms to have and not to have the property through a sound and complete equational axiomatization. Specifically, we give examples of B-terms which have the property and show that there are infinitely many B-terms which do not have the property. Also, we introduce a canonical representation of B-terms that is useful to detect cycles, or equivalently, to prove the property, with an efficient algorithm.

1 Introduction

The 'bluebird' combinator $B = \lambda f.\lambda g.\lambda x.f\ (g\ x)$ is well-known [10] as a bracketing combinator or composition operator, which has a principal type $(\alpha \to \beta) \to (\gamma \to \alpha) \to \gamma \to \beta$. A function $B\ f\ g$ (also written as $f \circ g$) takes a single argument x and returns the term $f\ (g\ x)$.

In the general case that g takes n arguments, the composition of f and g, defined by $\lambda x_1. \cdots \lambda x_n.f\ (g\ x_1\ \ldots\ x_n)$, can be expressed as $B^n\ f\ g$ where e^n is the n-fold composition $\underbrace{e \circ \cdots \circ e}_{n}$ of the function e, or equivalently given by $e^n\ x = \underbrace{e\ (\ldots (e}_{n}\ x))$ [1, Definition 2.1.9]. We call *n-argument composition* for the generalized composition represented by B^n.

Now we consider the 2-argument composition expressed as $B^2 = \lambda f.\lambda g.\lambda x.\lambda y.\ f\ (g\ x\ y)$. From the definition, we have $B^2 = B \circ B = B\ B\ B$. Note that function application is considered left-associative, that is, $f\ a\ b = (f\ a)\ b$. Thus B^2 is expressed as a term in which all applications nest to the left, never to the right. We call such terms *flat* [9]. We write $X_{(k)}$ for the flat term defined by $\underbrace{X\ X\ X\ \ldots\ X}_{k} = \underbrace{(\ldots((X\ X)\ X)\ldots)\ X}_{k}$. Using this notation, we can write $B^2 = B_{(3)}$.

$$B_{(1)}-B_{(2)}-B_{(3)}-B_{(4)}-B_{(5)}-B_{(6)} \overline{\qquad\qquad\qquad} B_{(7)}$$

$$B_{(9)} \overline{\qquad\qquad\qquad} B_{(8)}$$

where:
- $B_{(6)}\ (= B_{(10)} = B_{(14)} = \ldots)$
- $B_{(7)}\ (= B_{(11)} = B_{(15)} = \ldots)$
- $B_{(9)}\ (= B_{(13)} = B_{(17)} = \ldots)$
- $B_{(8)}\ (= B_{(12)} = B_{(16)} = \ldots)$

Figure 1 ρ-property of the B combinator.

Okasaki [9] investigated facts about flatness. For example, he shows that there is no universal combinator X that can represent any combinator by $X_{(k)}$ with some k. We shall delve into the case of $X = B$. Consider the n-argument composition operator B^n. We have already seen that B^2 can be written by the flat term $B_{(3)}$. For $n = 3$, we can also check $B^3 = B\ B\ B\ B\ B\ B\ B\ B = B_{(8)}$ by repeating β-reduction for $B_{(8)}\ f\ g\ x\ y\ z = f\ (g\ x\ y\ z)$. How about the 4-argument composition B^4? In fact, there is no integer k such that $B^4 = B_{(k)}$ with respect to $\beta\eta$-equality. Moreover, for any $n > 3$, there does not exist k such that $B^n = B_{(k)}$. This surprising fact is proved by a quite simple method; listing all $B_{(k)}$s for $k = 1, 2, \ldots$ and checking that none of them is equivalent to B^n. An easy computation gives $B_{(6)} = B_{(10)} = \lambda x.\lambda y.\lambda z.\lambda w.\lambda v.\ x\ (y\ z)\ (w\ v)$, and hence $B_{(i)} = B_{(i+4)}$ for every $i \geq 6$. Then, by computing $B_{(k)}$s only for $k \in \{1, 2, \ldots, 6\}$, we can check that $B_{(k)}$ is not $\beta\eta$-equivalent to B^n with $n > 3$ for $k \in \{1, 2, \ldots\}$. Thus we conclude that there is no integer k such that $B^n = B_{(k)}$.

This is the starting point of our research. We call ρ-*property* for this "periodicity" on combinatory terms. More precisely, we say that a combinator X has ρ-property if there exist two distinct integers i and j such that $X_{(i)} = X_{(j)}$. In this case, we have $X_{(i+k)} = X_{(j+k)}$ for any $k \geq 0$ (à la *finite monogenic semigroup* [7]). Fig. 1 shows a computation graph of $B_{(k)}$. The ρ-property is named after the shape of the graph.

This paper discusses the ρ-property of combinatory terms, particularly terms built from B alone. We call such terms B-*terms* and $\mathbf{CL}(B)$ denotes the set of all B-terms. For example, the B-term $B\ B$ enjoys the ρ-property with $(B\ B)_{(52)} = (B\ B)_{(32)}$ and so does $B\ (B\ B)$ with $(B\ (B\ B))_{(294)} = (B\ (B\ B))_{(258)}$ as reported in [8]. Several combinators other than B-terms can be found to enjoy the ρ-property, for example, $K = \lambda x.\lambda y.x$ and $C = \lambda x.\lambda y.\lambda z.\ x\ z\ y$ because of $K_{(3)} = K_{(1)}$ and $C_{(4)} = C_{(3)}$. They are less interesting in the sense that the cycle starts immediately and its size is very small, comparing with B-terms like $B\ B$ and $B\ (B\ B)$. As we will see later, $B\ (B\ (B\ (B\ (B\ (B\ B)))))(\equiv B^6\ B)$ has the ρ-property with the cycle of the size more than 3×10^{11} which starts after more than 2×10^{12} repetitive right applications. This is why the ρ-property of B-terms is intensively discussed in the present paper.

The contributions of the paper are two-fold. One is to give a characterization of $\mathbf{CL}(B)$ (Section 3) and another is to provide a sufficient condition for the ρ-property and anti-ρ-property of B-terms (Section 4). In the former, we introduce a canonical representation of B-terms and establish a sound and complete equational axiomatization for $\mathbf{CL}(B)$. In the latter, the ρ-property of B^nB with $n \leq 6$ is shown with an efficient algorithm and the anti-ρ-property for B-terms of particular forms is proved.

2 ρ-property of terms

The ρ-property of combinator X is that $X_{(i)} = X_{(j)}$ holds for some $i > j \geq 1$. We adopt $\beta\eta$-equality of corresponding λ-terms for the equality of combinatory terms in this paper. We could use other equality, for example, induced by the axioms of combinatory logic. The choice of equality is not essential here, e.g., $B_{(9)}$ and $B_{(13)}$ are equal even up to the combinatory axiom of B, as well as $\beta\eta$-equality. Furthermore, for simplicity, we only deal with the case

$$\rho(B^0 B) = (6, 4) \qquad\qquad \rho(B^4 B) = (191206, 431453)$$
$$\rho(B^1 B) = (32, 20) \qquad\qquad \rho(B^5 B) = (766241307, 234444571)$$
$$\rho(B^2 B) = (258, 36) \qquad\qquad \rho(B^6 B) = (2641033883877, 339020201163)$$
$$\rho(B^3 B) = (4240, 5796)$$

▨ **Figure 2** ρ-property of B-terms in a particular form.

where $X_{(n)}$ is normalizable for all n. If $X_{(n)}$ is not normalizable, it is much more difficult to check equivalence with the other terms. This restriction does not affect results of the paper because all B-terms are normalizing.

Let us write $\rho(X) = (i, j)$ if a combinator X has the ρ-property due to $X_{(i)} = X_{(i+j)}$ with minimum positive integers i and j. For example, we have $\rho(B) = (6, 4)$, $\rho(C) = (3, 1)$, $\rho(K) = (1, 2)$ and $\rho(I) = (1, 1)$. Besides them, several combinators introduced in Smullyan's book [10] have the ρ-property:

$$\rho(D) = (32, 20) \qquad\qquad \text{where } D = \lambda x.\lambda y.\lambda z.\lambda w.x\ y\ (z\ w)$$
$$\rho(F) = (3, 1) \qquad\qquad \text{where } F = \lambda x.\lambda y.\lambda z.z\ y\ x$$
$$\rho(R) = (3, 1) \qquad\qquad \text{where } R = \lambda x.\lambda y.\lambda z.y\ z\ x$$
$$\rho(T) = (2, 1) \qquad\qquad \text{where } T = \lambda x.\lambda y.y\ x$$
$$\rho(V) = (3, 1) \qquad\qquad \text{where } V = \lambda x.\lambda y.\lambda z.z\ x\ y.$$

Except the B and D $(= B\ B)$ combinators, the property is 'trivial' in the sense that the loop starts early and the size of cycle is very small.

On the other hand, the combinators $S = \lambda x.\lambda y.\lambda z.x\ z\ (y\ z)$ and $O = \lambda x.\lambda y.y\ (x\ y)$ in the book do not have the ρ-property for reason (A), which is illustrated by

$$S_{(2n+1)} = \lambda x.\lambda y.\underbrace{x\ y\ (x\ y\ (\ldots(x\ y}_{n}\ (\lambda z.x\ z\ (y\ z)))\ldots)),$$

$$O_{(n+1)} = \lambda x.\underbrace{x\ (x\ (\ldots(x}_{n}\ (\lambda y.y\ (x\ y)).$$

The definition of the ρ-property is naturally extended from single combinators to terms obtained by combining several combinators. We found by computation that several B-terms, built from the B combinator alone, have a nontrivial ρ-property as shown in Fig. 2. The detail will be shown in Section 4.

3 Checking equivalence of B-terms

The set of all B-terms, $\mathbf{CL}(B)$, is closed under application by definition, that is, the repetitive right application of a B-term always generates a sequence of B-terms. Hence, the ρ-property can be decided by checking 'equivalence' among generated B-terms, where the equivalence should be checked through $\beta\eta$-equivalence of their corresponding λ-terms in accordance with the definition of the ρ-property. It would be useful if we have a fast algorithm for deciding equivalence over B-terms.

In this section, we give a characterization of the B-terms to efficiently decide their equivalence. We introduce a method for deciding equivalence of B-terms without calculating the corresponding λ-terms. To this end, we first investigate equivalence over B-terms with

$$B \ x \ y \ z = x \ (y \ z) \tag{B1}$$

$$B \ (B \ x \ y) = B \ (B \ x) \ (B \ y) \tag{B2}$$

$$B \ B \ (B \ x) = B \ (B \ (B \ x)) \ B \tag{B3}$$

■ **Figure 3** Equational axiomatization for B-terms

examples and then present an equation system as a characterization of B-terms so as to decide equivalence between two B-terms. Based on the equation system, we introduce a canonical representation of B-terms. The representation makes it easy to observe the growth caused by repetitive right application of B-terms, which will be later used for proving the anti-ρ-property of B^2. We believe that this representation will be helpful to prove the ρ-property or the anti-ρ-property for the other B-terms.

3.1 Equivalence over B-terms

Two B-terms are said *equivalent* if their corresponding λ-terms are $\beta\eta$-equivalent. For instance, $B \ B \ (B \ B)$ and $B \ (B \ B) \ B \ B$ are equivalent. This can be easily shown by the definition $B \ x \ y \ z = x \ (y \ z)$. For another (non-trivial) instance, $B \ B \ (B \ B)$ and $B \ (B \ (B \ B)) \ B$ are equivalent. This is illustrated by the fact that they are equivalent to $\lambda x.\lambda y.\lambda z.\lambda w.\lambda v.x \ (y \ z) \ (w \ v)$ where B is replaced with $\lambda x.\lambda y.\lambda z. \ x \ (y \ z)$ or the other way around at the $=_\beta$ equation. Similarly, we cannot show equivalence between two B-terms, $B \ (B \ B) \ (B \ B)$ and $B \ (B \ B \ B)$, without long calculation. This kind of equality makes it hard to investigate the ρ-property of B-terms. To solve this annoying issue, we will later introduce a canonical representation of B-terms.

3.2 Equational axiomatization for B-terms

Equality between two B-terms can be effectively decided by an equation system. Figure 3 shows a sound and complete equation system as described in the following theorem.

▶ **Theorem 1.** *Two B-terms are $\beta\eta$-equivalent if and only if their equality is derived by equations (B1), (B2), and (B3).*

The proof of the "if" part, which corresponds to the soundness of the equation system (B1), (B2), and (B3), is given here. We will later prove the "only if" part with the uniqueness of the canonical representation of B-terms.

Proof. Equation (B1) is immediate from the definition of B. Equations (B2) and (B3) are shown by

$$
\begin{aligned}
B \ (B \ e_1 \ e_2) &= \lambda x.\lambda y. \ B \ (B \ e_1 \ e_2) \ x \ y \\
&= \lambda x.\lambda y. \ B \ e_1 \ e_2 \ (x \ y) \\
&= \lambda x.\lambda y. \ e_1 \ (e_2 \ (x \ y)) \\
&= \lambda x.\lambda y. \ e_1 \ (B \ e_2 \ x \ y) \\
&= \lambda x. \ B \ e_1 \ (B \ e_2 \ x) \\
&= B \ (B \ e_1) \ (B \ e_2)
\end{aligned}
\qquad
\begin{aligned}
B \ B \ (B \ e_1) &= \lambda x. \ B \ B \ (B \ e_1) \ x \\
&= \lambda x. \ B \ (B \ e_1 \ x) \\
&= \lambda x.\lambda y.\lambda z. \ B \ e_1 \ x(y \ z) \\
&= \lambda x.\lambda y.\lambda z. \ e_1 \ (x \ (y \ z)) \\
&= \lambda x.\lambda y.\lambda z. \ e_1 \ (B \ x \ y \ z) \\
&= \lambda x.\lambda y. \ B \ e_1 \ (B \ x \ y) \\
&= \lambda x. \ B \ (B \ e_1) \ (B \ x) \\
&= B \ (B \ (B \ e_1)) \ B
\end{aligned}
$$

where the α-renaming is implicitly used.   ◀

Equation (B2) has been employed by Statman [12] to show that no $B\omega$-term can be a fixed-point combinator where $\omega = \lambda x.x\ x$. This equation exposes an interesting feature of the B combinator. Write equation (B2) as

$$B\ (e_1 \circ e_2) = (B\ e_1) \circ (B\ e_2) \tag{B2'}$$

by replacing every B combinator with $\circ$ infix operator if it has exactly two arguments. The equation is a distributive law of B over $\circ$, which will be used to obtain the canonical representation of B-terms. Equation (B3) is also used for the same purpose as the form of

$$B \circ (B\ e_1) = (B\ (B\ e_1)) \circ B. \tag{B3'}$$

We also have a natural equation $B\ e_1\ (B\ e_2\ e_3) = B\ (B\ e_1\ e_2)\ e_3$ which represents associativity of function composition, i.e., $e_1 \circ (e_2 \circ e_3) = (e_1 \circ e_2) \circ e_3$. This is shown with equations (B1) and (B2) by

$$B\ e_1\ (B\ e_2\ e_3) = B\ (B\ e_1)\ (B\ e_2)\ e_3 = B\ (B\ e_1\ e_2)\ e_3.$$

3.3 Canonical representation of B-terms

To decide equality between two B-terms, it does not suffice to compute their normal forms under the definition of B, $B\ x\ y\ z \to x\ (y\ z)$. This is because two distinct normal forms may be equal up to $\beta\eta$-equivalence, e.g., $B\ B\ (B\ B)$ and $B\ (B\ (B\ B))\ B$. We introduce a canonical representation of B-terms, which makes it easy to check equivalence of B-terms. We will eventually find that for any B-term e there exists a unique finite non-empty weakly-decreasing sequence of non-negative integers $n_1 \geq n_2 \geq \cdots \geq n_k$ such that e is equivalent to $(B^{n_1}B) \circ (B^{n_2}B) \circ \cdots \circ (B^{n_k}B)$. Ignoring the inequality condition gives *polynomials* introduced by Statman [12]. We will use these *decreasing polynomials* for our canonical representation as presented later. A similar result is found in [4].

First, we explain how this canonical form is obtained from a B-term. We only need to consider B-terms in which every B has at most two arguments. One can easily reduce the arguments of B to less than three by repeatedly rewriting occurrences of $B\ e_1\ e_2\ e_3\ e_4\ \ldots\ e_n$ into $e_1\ (e_2\ e_3)\ e_4\ \ldots\ e_n$. The rewriting procedure always terminates because it reduces the number of B. Thus, every B-term in $\mathbf{CL}(B)$ is equivalent to a B-term built by the syntax

$$e ::= B \ \mid\ B\ e \ \mid\ e \circ e \tag{1}$$

where $e_1 \circ e_2$ denotes $B\ e_1\ e_2$. We prefer to use the infix operator $\circ$ instead of B that has two arguments because associativity of B, that is, $B\ e_1\ (B\ e_2\ e_3) = B\ (B\ e_1\ e_2)\ e_3$ can be implicitly assumed. This simplifies the further discussion on B-terms. We will deal with only B-terms in syntax (1) from now on. The $\circ$ operator has a lower precedence than application in this paper, e.g., terms $B\ B \circ B$ and $B \circ B\ B$ represent $(B\ B) \circ B$ and $B \circ (B\ B)$, respectively.

The syntactic restriction by (1) does not suffice to proffer a canonical representation of B-terms. For example, both of the two B-terms $B \circ B\ B$ and $B\ (B\ B) \circ B$ are given in the form of (1), but we can see they are equivalent using (B3').

A *polynomial form of B-terms* is obtained by putting a restriction on the syntax so that no B combinator occurs outside of the $\circ$ operator while syntax (1) allows the B combinators and the $\circ$ operators to occur in an arbitrary position. The restricted syntax is given as

$$e ::= e_B \ \mid\ e \circ e \qquad e_B ::= B \ \mid B\ e_B$$

where terms in e_B have a form of $B(\dots(B(B\ B))\dots)$, that is $B^n B$ with some n, called *monomial*. The syntax can be simply rewritten into $e ::= B^n B \mid e \circ e$, which is called *polynomial*.

▶ **Definition 2.** A B-term $B^n B$ is called *monomial*. A *polynomial* is a B-term given in the form of

$$(B^{n_1} B) \circ (B^{n_2} B) \circ \cdots \circ (B^{n_k} B)$$

where $k > 0$ and $n_1, \dots, n_k \geq 0$ are integers. In particular, a polynomial is called *decreasing* when $n_1 \geq n_2 \geq \cdots \geq n_k$. The *length* of a polynomial P is a number of monomials in P, i.e., the length of the polynomial above is k. The numbers $n_1, n_2, \dots, n_k$ are called *degrees*.

In the rest of this subsection, we prove that for any B-term e there exists a unique decreasing polynomial equivalent to e. First, we show that e has an equivalent polynomial.

▶ **Lemma 3** ([12]). *For any B-term e, there exists a polynomial equivalent to e.*

Proof. We prove the statement by induction on the structure of e. In the case of $e \equiv B$, the term itself is polynomial. In the case of $e \equiv B\ e_1$, assume that e_1 has equivalent polynomial $(B^{n_1} B) \circ (B^{n_2} B) \circ \cdots \circ (B^{n_k} B)$. Repeatedly applying equation (B2') to $B\ e_1$, we obtain a polynomial equivalent to $B\ e_1$ as $(B^{n_1+1} B) \circ (B^{n_2+1} B) \circ \cdots \circ (B^{n_k+1} B)$. In the case of $e \equiv e_1 \circ e_2$, assume that e_1 and e_2 have equivalent polynomials P_1 and P_2, respectively. A polynomial equivalent to e is given by $P_1 \circ P_2$. ◀

Next, we show that for any polynomial P there exists a decreasing polynomial equivalent to P. A key equation of the proof is

$$(B^m B) \circ (B^n B) = (B^{n+1} B) \circ (B^m B) \quad \text{when } m < n, \tag{2}$$

which is shown by

$$
\begin{aligned}
(B^m B) \circ (B^n B) &= B^m (B \circ (B^{n-m} B)) \\
&= B^m (B \circ (B\ (B^{n-m-1} B))) \\
&= B^m ((B(B(B^{n-m-1} B))) \circ B) \\
&= (B^{n+1} B) \circ (B^m B)
\end{aligned}
$$

using equations (B2') and (B3').

▶ **Lemma 4.** *Any polynomial P has an equivalent decreasing polynomial P' such that*
- *the length of P and P' are equal, and*
- *the lowest degrees of P and P' are equal.*

Proof. We prove the statement by induction on the length of P. When the length is 1, that is, P is a monomial, P itself is decreasing and the statement holds. When the length k of P is greater than 1, take P_1 such that $P \equiv P_1 \circ (B^n B)$. From the induction hypothesis, there exists a decreasing polynomial $P_1' \equiv (B^{n_1} B) \circ (B^{n_2} B) \circ \cdots \circ (B^{n_{k-1}} B)$ equivalent to P_1, and the lowest degree of P_1 is n_{k-1}. If $n_{k-1} \geq n$, then $P' \equiv P_1' \circ (B^n\ B)$ is decreasing and equivalent to P. Since the lowest degrees of P and P' are n, the statement holds. If $n_{k-1} < n$, P is equivalent to

$$(B^{n_1}\ B) \circ \cdots \circ (B^{n_{k-1}} B) \circ (B^n B) = (B^{n_1} B) \circ \cdots \circ (B^{n+1} B) \circ (B^{n_{k-1}} B)$$

due to equation (2). Putting the last term as $P_2 \circ (B^{n_{k-1}}B)$, the length of P_2 is $k-1$ and the lowest degree of P_2 is greater than or equal to n_{k-1}. From the induction hypothesis, P_2 has an equivalent decreasing polynomial P_2' of length $k-1$ and the lowest degree of P_2' greater than or equal to n_{k-1}. Thereby we obtain a decreasing polynomial $P_2' \circ (B^{n_{k-1}}B)$ equivalent to P and the statement holds. ◀

▶ **Example 5.** Consider a B-term $e = B\ (B\ B\ B)\ (B\ B)\ B$. First, applying equation (B1),

$$e = B\ (B\ B\ B)\ (B\ B)\ (B\ B) = B\ B\ B\ (B\ B\ (B\ B)) = B\ (B\ (B\ B\ (B\ B)))$$

so that every B has at most two arguments. Then replace each B to the infix $\circ$ operator if it has two arguments and obtain $B\ (B\ (B \circ (B\ B)))$ Applying equation (B2'), we have

$$\begin{aligned}
B\ (B\ (B \circ (B\ B))) &= B\ ((B\ B) \circ (B\ (B\ B))) \\
&= (B\ (B\ B)) \circ (B\ (B\ (B\ B))) \\
&= (B^2 B) \circ (B^3 B).
\end{aligned}$$

Applying equation (2), we obtain the decreasing polynomial $(B^4 B) \circ (B^2 B)$ equivalent to e.

Every B-term has at least one equivalent decreasing polynomial as shown so far. To conclude this subsection, we show the uniqueness of decreasing polynomial equivalent to any B-term, that is, every B-term e has no two distinct decreasing polynomials equivalent to e.

The proof is based on the idea that B-terms correspond to unlabeled binary trees. Let M be a term which is constructed from variables $x_1, \ldots, x_k$ and their applications. Then we can show that if the λ-term $\lambda x_1. \ldots \lambda x_k.\ M$ is in $\mathbf{CL}(B)$, then M is obtained by putting parentheses to some positions in the sequence $x_1\ \ldots\ x_k$. More precisely, we have the following lemma.

▶ **Lemma 6.** *Every λ-term in $\mathbf{CL}(B)$ is $\beta\eta$-equivalent to a λ-term of the form $\lambda x_1. \ldots \lambda x_k.\ M$ with some $k > 2$ where M satisfies the following two conditions: (1) M consists of only the variables $x_1, \ldots, x_k$ and their applications, and (2) for every subterm of M which is in the form of $M_1\ M_2$, if M_1 has a variable x_i, then M_2 does not have any variable x_j with $j \leq i$.*

Proof. By the structural induction of B-terms. ◀

From this lemma, we see that we do not need to specify variables in M and we can simply write like $\star\ \star\ (\star\ \star) =\ x_1\ x_2\ (x_3\ x_4)$. Formally speaking, every λ-term in $\mathbf{CL}(B)$ uniquely corresponds to a term built from $\star$ alone by the map $(\lambda x_1. \ldots \lambda x_k.\ M) \mapsto M[\star/x_1, \ldots, \star/x_k]$. We say an unlabeled binary tree (or simply, binary tree) for a term built from $\star$ alone since every term built from $\star$ alone can be seen as an unlabeled binary tree. (A term $\star$ corresponds to a leaf and $t_1\ t_2$ corresponds to the tree with left subtree t_1 and right subtree t_2.) To specify the applications in binary trees, we write $\langle t_1, t_2 \rangle$ for the application $t_1\ t_2$. For example, B-terms $B = \lambda x.\lambda y.\lambda z.\ x\ (y\ z)$ and $B\ B = \lambda x.\lambda y.\lambda z.\lambda w.\ x\ y\ (z\ w)$ are represented by $\langle \star, \langle \star, \star \rangle \rangle$ and $\langle \langle \star, \star \rangle, \langle \star, \star \rangle \rangle$, respectively.

We will present an algorithm for constructing the corresponding decreasing polynomial from a given binary tree. First let us define a function $\mathcal{L}_i$ with integer i which maps binary trees to lists of integers:

$$\mathcal{L}_i(\star) = [\ ] \qquad\qquad \mathcal{L}_i(\langle t_1, t_2 \rangle) = \mathcal{L}_{i+\|t_1\|}(t_2) +\!\!+ \mathcal{L}_i(t_1) +\!\!+ [i]$$

where $+\!\!+$ concatenates two lists and $\|t\|$ denotes a number of leaves. For example, $\mathcal{L}_0(\langle \langle \star, \star \rangle, \langle \star, \star \rangle \rangle) = [2, 0, 0]$ and $\mathcal{L}_1(\langle \langle \star, \langle \star, \star \rangle \rangle, \langle \star, \langle \star, \star \rangle \rangle \rangle) = [4, 4, 2, 1, 1]$. Informally, the

$\mathcal{L}_i$ function returns a list of integers which is obtained by labeling both leaves and nodes in the following steps. First each leaf of a given tree is labeled by $i, i+1, i+2, \ldots$ in left-to-right order. Then each binary node of the tree is labeled by the same label as its leftmost descendant leaf. The $\mathcal{L}_i$ functions return a list of only node labels in decreasing order. The length of the list equals the number of nodes, that is, smaller by one than the number of variables in the λ-term.

We define a function $\mathcal{L}$ which takes a binary tree t and returns a list of non-negative integers in $\mathcal{L}_{-1}(t)$, that is, the list obtained by excluding trailing all -1's in $\mathcal{L}_{-1}(t)$. Note that by excluding the label -1's it may happen to be $\mathcal{L}(t) = \mathcal{L}(t')$ for two distinct binary trees t and t' even though the $\mathcal{L}_i$ function is injective. However, those binary trees t and t' must be 'η-equivalent' in terms of the corresponding λ-terms.

The following lemma claims that the $\mathcal{L}$ function computes a list of degrees of a decreasing polynomial corresponding to a given λ-term.

▶ **Lemma 7.** *A decreasing polynomial* $(B^{n_1}B) \circ (B^{n_2}B) \circ \cdots \circ (B^{n_k}B)$ *is $\beta\eta$-equivalent to a λ-term $e \in \mathbf{CL}(B)$ corresponding a binary tree t such that $\mathcal{L}(t) = [n_1, n_2, \ldots, n_k]$.*

Proof. We prove the statement by induction on the length of the polynomial P.

When $P \equiv B^n B$ with $n \geq 0$, it is found to be equivalent to the λ-term

$$\lambda x_1.\lambda x_2.\lambda x_3.\ldots.\lambda x_{n+1}.\lambda x_{n+2}.\lambda x_{n+3}.\ x_1\ x_2\ x_3\ \ldots\ x_{n+1}\ (x_{n+2}\ x_{n+3})$$

by induction on n. This λ-term corresponds to a binary tree $t = \langle\langle \ldots \langle\langle \star, \star \rangle, \star \rangle, \ldots, \star \rangle, \langle \star, \star \rangle \rangle$.
$$\underbrace{}_{n \text{ leaves}}$$

Then we have $\mathcal{L}(t) = [n]$ holds from $\mathcal{L}_{-1}(t) = [n, \underbrace{-1, -1, \ldots, -1}_{n+1}]$.

When $P \equiv P' \circ (B^n B)$ with $P' \equiv (B^{n_1}B) \circ \cdots \circ (B^{n_k}B)$, $k \geq 1$ and $n_1 \geq \cdots \geq n_k \geq n \geq 0$, there exists a λ-term $\beta\eta$-equivalent to P' corresponding a binary tree t' such that $\mathcal{L}(t') = [n_1, \ldots, n_k]$ from the induction hypothesis. The binary tree t' must have the form of $\langle\langle\langle \ldots \langle\langle \star, \star \rangle, \star \rangle, \ldots, \star \rangle, t_1 \rangle, \ldots, t_m \rangle$ with $m \geq 1$ and some trees $t_1, \ldots, t_m$, otherwise $\mathcal{L}(t')$
$$\underbrace{}_{n_k \text{ leaves}}$$
would contain an integer smaller than n_k. From the definition of $\mathcal{L}$ and $\mathcal{L}_i$, we have

$$\mathcal{L}(t') = \mathcal{L}_{s_m}(t_m) \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} \mathcal{L}_{s_1}(t_1) \tag{3}$$

where $s_j = n_k + 1 + \sum_{i=1}^{j-1} \|t_i\|$. Additionally, the structure of t' implies $P' = \lambda x_1.\ldots.\lambda x_l.$ $x_1\ x_2\ \ldots\ x_{n_k+1}\ e_1 \ldots e_m$ where e_i corresponds to a binary tree t_i for $i = 1, \ldots, m$. From $B^n\ B = \lambda y_1.\ldots.\lambda y_{n+3}.\ y_1\ y_2 \ldots y_{n+1}\ (y_{n+2}\ y_{n+3})$, we compute a λ-term $\beta\eta$-equivalent to $P \equiv P' \circ (B^n B)$ by

$$P = \lambda x.\ P'(B^n B\ x)$$
$$= \lambda x.\ (\lambda x_1.\ldots.\lambda x_l.\ x_1\ x_2 \ldots x_{n_k+1}\ e_1\ \ldots\ e_m)$$
$$\qquad (\lambda y_2.\ldots.\lambda y_{n+3}.\ x\ y_2 \ldots y_{n+1}\ (y_{n+2}\ y_{n+3}))$$
$$= \lambda x.\lambda x_2.\ldots.\lambda x_l.\ (\lambda y_2.\ldots.\lambda y_{n+3}.\ x\ y_2 \ldots y_{n+1}\ (y_{n+2}\ y_{n+3}))\ x_2 \ldots x_{n_k+1}\ e_1 \ldots e_m$$
$$= \lambda x.\lambda x_2.\ldots.\lambda x_l.$$
$$\qquad (\lambda y_{n+1}.\lambda y_{n+2}.\lambda y_{n+3}.\ x\ x_2 \ldots x_n\ y_{n+1}\ (y_{n+2}\ y_{n+3}))\ x_{n+1} \ldots x_{n_k+1}\ e_1 \ldots e_m$$

where $n_k \geq n$ is taken into account. We split into four cases: (i) $n_k = n$ and $m = 1$, (ii) $n_k = n$ and $m > 1$, (iii) $n_k = n + 1$, and (iv) $n_k > n + 1$. In the case (i) where $n_k = n$ and $m = 1$, we have

$$P = \lambda x.\lambda x_2.\ldots.\lambda x_l.\lambda y_{n+3}.\ x\ x_2 \ldots x_n\ x_{n+1}\ (e_1\ y_{n+3}).$$

whose corresponding binary tree t is $\langle\langle\ldots\langle\langle\star,\underbrace{\star\rangle,\star\rangle,\ldots,\star}_{n\text{ leaves}}\rangle,\langle t_1,\star\rangle\rangle$. From equation (3), $\mathcal{L}(t) = \mathcal{L}_{n+1}(t_1) + [n+1] = \mathcal{L}(t') + [n+1] = [n_1,\ldots,n_k,n+1]$, thus the statement holds. In the case (ii) where $n_k = n$ and $m > 1$, we have

$$P = \lambda x.\lambda x_2.\ldots.\lambda x_l.\ x\ x_2\ldots x_n\ x_{n+1}\ (e_1\ e_2)\ e_3\ldots e_m.$$

whose corresponding binary tree t is $\langle\langle\langle\ldots\langle\langle\star,\underbrace{\star\rangle,\star\rangle,\ldots,\star}_{n\text{ leaves}}\rangle,\langle t_1,t_2\rangle,t_3\rangle,\ldots,t_m\rangle$. Hence, $\mathcal{L}(t) = \mathcal{L}(t') + [n+1]$ holds again from equation (3). In the case (iii) where $n_k = n+1$, we have

$$P = \lambda x.\lambda x_2.\ldots.\lambda x_l.\ x\ x_2\ldots x_n\ x_{n+1}\ (x_{n+2}\ e_1)\ e_2\ldots e_m,\ \text{or}$$

whose corresponding binary tree t is $\langle\langle\langle\ldots\langle\langle\star,\underbrace{\star\rangle,\star\rangle,\ldots,\star}_{n\text{ leaves}}\rangle,\langle\star,t_1\rangle,t_2\rangle,\ldots,t_m\rangle$. Hence, $\mathcal{L}(t) = \mathcal{L}(t') + [n+1]$ holds from equation (3). In the case (iv) where $n_k \geq n+2$, we have

$$P = \lambda x.\lambda x_2.\ldots.\lambda x_l.\ x\ x_2\ldots x_n\ x_{n+1}\ (x_{n+2}\ x_{n+3})\ \ldots\ e_1\ldots e_m,$$

whose corresponding binary tree t is $\langle\langle\langle\ldots\langle\langle\star,\underbrace{\star\rangle,\star\rangle,\ldots,\star}_{n\text{ leaves}}\rangle,\langle\star,\star\rangle,\ldots,t_1\rangle,\ldots,t_m\rangle$. Hence, $\mathcal{L}(t) = \mathcal{L}(t') + [n+1]$ holds from equation (3). ◀

▶ **Example 8.** A λ-term $\lambda x_1.\lambda x_2.\lambda x_3.\lambda x_4.\lambda x_5.\lambda x_6.\lambda x_7.\lambda x_8.\ x_1\ (x_2\ x_3)\ (x_4\ x_5\ x_6\ (x_7\ x_8))$ is $\beta\eta$-equivalent to $(B^5\ B) \circ (B^2\ B) \circ (B^2\ B) \circ (B^2\ B) \circ (B^0\ B)$ because its corresponding binary tree $t = \langle\langle\star,\langle\star,\star\rangle\rangle,\langle\langle\langle\star,\star\rangle,\star\rangle,\langle\star,\star\rangle\rangle\rangle$ satisfies $\mathcal{L}(t) = [5,2,2,2,0]$.

The previous lemmas immediately conclude the uniqueness of decreasing polynomials for B-terms shown in the following theorem.

▶ **Theorem 9.** *Every B-term e has a unique decreasing polynomial.*

Proof. For any given B-term e, we can find a decreasing polynomial for e from Lemma 3 and Lemma 4. Since every decreasing polynomial corresponds to only one binary tree (and since every B-term also corresponds to only one binary tree up to η-equivalence) from Lemma 7, the present statement holds. ◀

This theorem implies that the decreasing polynomial of B-terms can be used as their *canonical representation*, which is effectively derived as shown in Lemma 3 and Lemma 4.

As a corollary of the theorem, we can show the "only if" statement of Theorem 1, which corresponds to the completeness of the equation system.

Proof. Let e_1 and e_2 be equivalent B-terms, that is, their λ-terms are $\beta\eta$-equivalent. From Theorem 9, their decreasing polynomials are the same. Since the decreasing polynomial is derived from e_1 and e_2 by equations (B1), (B2), and (B3) according to the proofs of Lemma 3 and Lemma 4, equivalence between e_1 and e_2 is also derived from these equations. ◀

4 Results on the ρ-property of B-terms

We investigate the ρ-property of concrete B-terms, some of which have the property and others do not. For B-terms having the ρ-property, we introduce an efficient implementation to compute the entry point and the size of the cycle. For B-terms not having the ρ-property, we give a proof why they do not have.

4.1 B-terms having the ρ-property

As shown in Section 2, we can check that B-terms equivalent to $B^n B$ with $n \leq 6$ have the ρ-property by computing $(B^n B)_{(i)}$ for each i. However, it is not easy to check it by computer without an efficient implementation because we should compute all $(B^6 B)_{(i)}$ with $i \leq 2980054085040$ ($= 2641033883877 + 339020201163$) to know that $\rho(B^6 B) = (2641033883877, 339020201163)$. A naive implementation which computes terms of $(B^6 B)_{(i)}$ for all i and stores all of them has no hope to detect the ρ-property.

We introduce an efficient procedure to find the ρ-property of B-terms which can successfully compute $\rho(B^6 B)$. The procedure is based on two orthogonal ideas, Floyd's cycle-finding algorithm [6] and an efficient right application algorithm over decreasing polynomials presented in Section 3.3.

The first idea, Floyd's cycle-finding algorithm (also called the tortoise and the hare algorithm), enables us to detect the cycle with a constant memory usage, that is, the history of all terms $X_{(i)}$ does not need to be stored to check the ρ-property of the X combinator. The key of this algorithm is the fact that there are two distinct integers i and j with $X_{(i)} = X_{(j)}$ if and only if there is an integer m with $X_{(m)} = X_{(2m)}$, where the latter requires to compare $X_{(i)}$ and $X_{(2i)}$ from smaller i and store only these two terms for the next comparison between $X_{(i+1)} = X_{(i)} X$ and $X_{(2i+2)} = X_{(2i)} X X$ when $X_{(i)} \neq X_{(2i)}$. The following procedure computes the entry point and the size of the cycle if X has the ρ-property.
1. Find the smallest m such that $X_{(m)} = X_{(2m)}$.
2. Find the smallest k such that $X_{(k)} = X_{(m+k)}$.
3. Find the smallest $0 < c \leq k$ such that $X_{(m)} = X_{(m+c)}$. If not found, put $c = m$.
After this procedure, we find $\rho(X) = (k, c)$. The third step can be run in parallel during the second one. See [6, exercise 3.1.6] for the detail. One could use slightly more (possibly) efficient algorithms by Brent [3] and Gosper [2, item 132] for cycle detection.

Efficient cycle-finding algorithms do not suffice to compute $\rho(B^6 B)$. Only with the idea above running on a laptop (1.7 GHz Intel Core i7 / 8GB of memory), it takes about 2 hours even for $\rho(B^5 B)$ and fails to compute $\rho(B^6 B)$ with an out-of-memory error.

The second idea enables us to efficiently compute $X_{(i+1)}$ from $X_{(i)}$ for B-terms X. The key of this algorithm is to use the canonical representation of $X_{(i)}$, that is a decreasing polynomial, and directly compute the canonical representation of $X_{(i+1)}$ from that of $X_{(i)}$. Additionally, the canonical representation enables us to quickly decide equivalence which is required many times to find the cycle. It takes time just proportional to their lengths. If λ-terms are used for finding the cycle, both application and deciding equivalence require much more complicated computation. Our implementation based on these two ideas computes $\rho(B^5 B)$ and $\rho(B^6 B)$ in 10 minutes and 59 days (!), respectively.

For two given decreasing polynomials P_1 and P_2, we show how a decreasing polynomial P equivalent to $(P_1 \ P_2)$ can be obtained. The method is based on the following lemma about application of one B-term to another B-term.

▶ **Lemma 10.** *For B-terms e_1 and e_2, there exists $k \geq 0$ such that $e_1 \circ (B \ e_2) = B \ (e_1 \ e_2) \circ B^k$.*

Proof. Let P_1 be a decreasing polynomial equivalent to e_1. We prove the statement by case analysis on the maximum degree in P_1. When the maximum degree is 0, we can take $k' \geq 1$ such that $P_1 \equiv \underbrace{B \circ \cdots \circ B}_{k'} = B^{k'}$. Then,

$$e_1 \circ (B \ e_2) = \underbrace{B \circ \cdots \circ B}_{k'} \circ (B \ e_2) = (B^{k'+1} e_2) \circ \underbrace{B \circ \cdots \circ B}_{k'} = B \ (e_1 \ e_2) \circ B^{k'}$$

where equation (B3') is used k' times in the second equation. Therefore the statement holds by taking $k = k'$. When the maximum degree is greater than 0, we can take a decreasing polynomial P' for a B-term and $k' \geq 0$ such that $P_1 = (B\ P') \circ \underbrace{B \circ \cdots \circ B}_{k'} = (B\ P') \circ B^{k'}$

due to equation (B2'). Then,

$$
\begin{aligned}
e_1 \circ (B\ e_2) &= (B\ P') \circ \underbrace{B \circ \cdots \circ B}_{k'} \circ (B\ e_2) \\
&= (B\ P') \circ (B^{k'+1} e_2) \circ \underbrace{B \circ \cdots \circ B}_{k'} \\
&= B\ (P' \circ (B^{k'} e_2)) \circ B^{k'} \\
&= B\ (B\ P'\ (B^{k'} e_2)) \circ B^{k'} \\
&= B\ (P_1\ e_2) \circ B^{k'} \\
&= B\ (e_1\ e_2) \circ B^{k'}.
\end{aligned}
$$

Therefore, the statement holds by taking $k = k'$. ◄

This lemma indicates that, from two decreasing polynomials P_1 and P_2, a decreasing polynomial P equivalent to $(P_1\ P_2)$ can be obtained in the following steps where L_1 and L_2 are lists of non-negative numbers as shown in Section 3.3 corresponding to P_1 and P_2.

1. Build P_2' by incrementing each degree of P_2 by 1, i.e., when $P_2 \equiv (B^{n_1} B) \circ \cdots \circ (B^{n_l} B)$, $P_2' \equiv (B^{n_1+1} B) \circ \cdots \circ (B^{n_l+1} B)$. In terms of the list representation, a list L_2' is built from L_2 by incrementing each value by 1.

2. Find a decreasing polynomial P_{12} corresponding to $P_1 \circ P_2'$ by equation (2). In terms of the list representation, a list L_{12} is constructed by appending L_1 and L_2' and repeatedly applying (2).

3. Obtain P by decrementing each degree of P_{12} after eliminating the trailing 0-degree units, i.e., when $P_{12} \equiv (B^{n_1} B) \circ \cdots \circ (B^{n_l} B) \circ (B^0 B) \circ \cdots \circ (B^0 B)$ with $n_1 \geq \cdots \geq n_l > 0$, $P \equiv (B^{n_1-1} B) \circ \cdots \circ (B^{n_l-1} B)$. In terms of the list representation, a list L is obtained from L_{12} by decrementing each value by 1 after removing trailing 0's.

In the first step, a decreasing polynomial P_2' equivalent to $B\ P_2$ is obtained. The second step yields a decreasing polynomial P_{12} for $P_1 \circ P_2' = P_1 \circ (B\ P_2)$. Since P_1 and P_2 are decreasing, it is easy to find P_{12} by repetitive application of equation (2) for each unit of P_2', à la insertion operation in insertion sort. In the final step, a polynomial P that satisfies $(B\ P) \circ B^k = P_{12}$ with some k is obtained. From Lemma 10 and the uniqueness of decreasing polynomials, P is equivalent to $(P_1\ P_2)$.

▶ **Example 11.** Let P_1 and P_2 be decreasing polynomials represented by lists $L_1 = [4, 1, 0]$ and $L_2 = [2, 0]$. Then a decreasing polynomial P equivalent to $(P_1\ P_2)$ is obtained as a list L in three steps:

1. A list $L_2' = [3, 1]$ is obtained from L_2 by incrementing each value by 1.
2. A decreasing list L_{12} is obtained from L_1 and L_2' by

$$
L_{12} = [4, 1, \underline{0, 3}, 1] = [4, \underline{1, 4}, 0, 1] = [\underline{4, 5}, 1, 0, 1] = [6, 4, 1, \underline{0, 1}] = [6, 4, \underline{1, 2}, 0] = [6, 4, 3, 1, 0]
$$

where equation (2) is applied in each underlined pair.

3. A list $L = [5, 3, 2, 0]$ is obtained from L_{12} as the result of the application by decrementing each value by 1 after removing trailing 0's.

The implementation based on the right application over decreasing polynomials is available at `https://github.com/ksk/Rho`. Note that the program does not terminate for the combinator which does not have the ρ-property. It will not help to decide if a combinator has the ρ-property. One might observe how the terms grow by repetitive right applications through running the program, though.

4.2 B-terms not having the ρ-property

We prove that the B-terms $(B^k B)^{(k+2)n}$ ($k \geq 0$, $n > 0$) do not have the ρ-property. For example, B-term $B^2 = B\ B\ B$, which is the case of $k = 0$ and $n = 1$, does not have the ρ-property. To this end, we show that the number of variables in the $\beta\eta$-normal form of $((B^k B)^{(k+2)n})_{(i)}$ is monotonically non-decreasing and that it implies the anti-ρ-property. Additionally, after proving that, we consider a sufficient condition not to have the ρ-property through the monotonicity.

First, we introduce some notations. Suppose that the $\beta\eta$-normal form of a B-term X is given by $\lambda x_1 \ldots \lambda x_n.\ x_1\ e_1\ \cdots\ e_k$ for some terms $e_1, \ldots, e_k$. Then we define $l(X) = n$ (the number of variables), $a(X) = k$ (the number of arguments of x_1), and $N_i(X) = e_i$ for $i = 1, \ldots, k$. For convinience, we define functions l, a, and N_i also for terms of form $Y = x\ e_1 \ldots e_k$ in the same mannar. That is, $l(Y)$ is the number of variables in Y, $a(Y) = k$, and $N_i(Y) = e_i$. Let X' be another B-term and suppose its $\beta\eta$-normal form is given by $\lambda x'_1 \ldots \lambda x'_{n'}.\ e'$ where e' does not have λ-abstractions. We can see $X\ X' = (\lambda x_1 \ldots \lambda x_n.\ x_1\ e_1\ \cdots\ e_k)\ X' = \lambda x_2 \ldots \lambda x_n.\ X'\ e_1\ \cdots\ e_k$ and from Lemma 6, its $\beta\eta$-normal form is

$$\begin{cases} \lambda x_2 \ldots \lambda x_n. \lambda x'_{k+1} \ldots \lambda x'_{n'}.\ e'[e_1/x'_1, \ldots, e_k/x'_k] & (k \leq n') \\ \lambda x_2 \ldots \lambda x_n.\ e'[e_1/x'_1, \ldots, e_{n'}/x'_{n'}]\ e_{n'+1}\ \cdots\ e_k & \text{(otherwise)}. \end{cases}$$

Here $e'[e_1/x'_1, \ldots, e_k/x'_k]$ is the term which is obtained by substituting $e_1, \ldots, e_k$ to the variables $x'_1, \ldots, x'_k$ in e' .

By simple computation with this fact, we get the following lemma:

▶ **Lemma 12.** *Let X and X' be B-terms. Then*

$$l(X\ X') = l(X) - 1 + \max\{l(X') - a(X),\ 0\}$$
$$a(X\ X') = a(X') + a(N_1(X)) + \max\{a(X) - l(X'),\ 0\}$$
$$N_1(X\ X') = \begin{cases} N_1(X')[N_2(X)/x'_2, \ldots, N_m(X)/x'_m] & \text{(if $N_1(X)$ is a variable)} \\ N_1(N_1(X)) & \text{(otherwise)} \end{cases}$$

where $m = \min\{l(N_1(X')),\ a(X)\}$.

The $\beta\eta$-normal form of $(B^k B)^{(k+2)n}$ is given by

$$\lambda x_1 \ldots \lambda x_{k+(k+2)n+2}.\ x_1\ x_2\ \cdots\ x_{k+1}\ (x_{k+2}\ x_{k+3}\ \cdots\ x_{k+(k+2)n+2}).$$

This is deduced from Lemma 7 since the binary tree corresponding to the above λ-term is $t = \langle\langle \ldots \langle\langle \underbrace{\star, \star\rangle, \star\rangle, \ldots, \star}_{k+1} \rangle, \langle \ldots \langle\langle \underbrace{\star, \star\rangle, \star\rangle, \ldots, \star}_{(k+2)n} \rangle\rangle$ and $\mathcal{L}(t) = [\underbrace{k, \ldots, k}_{(k+2)n}]$. Especially, we get $l((B^k B)^{(k+2)n}) = k + (k+2)n + 2$. In this section, we write $\langle \star, \star, \star, \ldots, \star\rangle$ for $\langle \ldots \langle\langle \star, \star\rangle, \star\rangle, \ldots, \star\rangle$ and identify B-terms with their corresponding binary trees.

To describe properties of $(B^k B)^{(k+2)n}$, we introduce a set $T_{k,n}$ which is closed under right application of $(B^k B)^{(k+2)n}$, that is, $T_{k,n}$ satisfies that "if $X \in T_{k,n}$ then $X\ (B^k B)^{(k+2)n} \in T_{k,n}$ holds". First we inductively define a set of terms $T'_{k,n}$ as follows:

1. $\star \in T'_{k,n}$
2. $\langle \star,\ s_1,\ \ldots,\ s_{(k+2)n} \rangle \in T'_{k,n}$ if $s_i = \star$ for each multiple i of $k + 2$ and $s_i \in T'_{k,n}$ for the others.

Then we define $T_{k,n}$ by $T_{k,n} = \left\{ \langle t_0,\ t_1,\ \ldots,\ t_{k+1} \rangle \ \middle|\ t_0,\ t_1, \ldots, t_{k+1} \in T'_{k,n} \right\}$. It is obvious that $(B^k B)^{(k+2)n} \in T_{k,n}$. Now we shall prove that $T_{k,n}$ is closed under right application of $(B^k B)^{(k+2)n}$.

▶ **Lemma 13.** *If $X \in T_{k,n}$ then $X\ (B^k B)^{(k+2)n} \in T_{k,n}$.*

Proof. From the definition of $T_{k,n}$, if $X \in T_{k,n}$ then X can be written in the form $\langle t_0,\ t_1,\ \ldots,\ t_{k+1} \rangle$ for some $t_0, \ldots, t_{k+1} \in T'_{k,n}$. In the case where $t_0 = \star$, we have $X\ (B^k B)^{(k+2)n} = \langle t_1,\ \ldots,\ t_{k+1},\ \langle \underbrace{\star,\ \ldots,\ \star}_{(k+2)n} \rangle \rangle \in T_{k,n}$. In the case where t_0 has the form of 2 in the definition of $T'_{k,n}$, then we have $X = \langle \star,\ s_1,\ \ldots,\ s_{(k+2)n},\ t_1,\ \ldots,\ t_{k+1} \rangle$ with $s_i = \star$ for each multiple i of $k + 2$ and $s_i \in T'_{k,n}$ for the others, hence

$$X\ (B^k B)^{(k+2)n} = \langle s_1,\ \ldots,\ s_{k+1},\ \langle s_{k+2},\ \ldots,\ s_{(k+2)n},\ t_1,\ \ldots,\ t_{k+1},\ \star \rangle \rangle.$$

We can easily see $s_1,\ \ldots,\ s_{k+1}$, and $\langle s_{k+2},\ \ldots,\ s_{(k+2)n},\ t_1,\ \ldots,\ t_{k+1},\ \star \rangle$ are in $T'_{k,n}$. ◀

From the definition of $T_{k,n}$, we can compute that $a(X)$ equals $k + 1$ or $(k+2)n + k + 1$ if $X \in T_{k,n}$. Particularly, we get the following:

▶ **Lemma 14.** *For any $X \in T_{k,n}$, $a(X) \le (k + 2)n + k + 1 = l((B^k B)^{(k+2)n}) - 1$.*

This lemma is crucial to show that the number of variables in $((B^k B)^{(k+2)n})_{(i)}$ is monotonically non-decreasing. Put $Z = (B^k B)^{(k+2)n}$ for short. Since $Z \in T_{k,n}$, we have $\{Z_{(i)} \mid i \ge 1\} \subset T_{k,n}$ by Lemma 13. Using Lemma 14, we can simplify Lemma 12 in the case where $X = Z_{(i)}$ and $X' = Z$ as follows:

$$l(Z_{(i+1)}) = l(Z_{(i)}) + (k + 2)n + k + 1 - a(Z_{(i)}) \tag{4}$$

$$a(Z_{(i+1)}) = a(N_1(Z_{(i)})) + k + 1 \tag{5}$$

$$N_1(Z_{(i+1)}) = \begin{cases} N_2(Z_{(i)}) & \text{(if } N_1(Z_{(i)}) \text{ is a variable)} \\ N_1(N_1(Z_{(i)})) & \text{(otherwise).} \end{cases} \tag{6}$$

By (4) and Lemma 14, we get $l(Z_{(i+1)}) \ge l(Z_{(i)})$.

To prove that Z does not have the ρ-property, it suffices to show the following:

▶ **Lemma 15.** *For any $i \ge 1$, there exists $j > i$ that satisfies $l(Z_{(j)}) > l(Z_{(i)})$.*

Proof. Suppose that there exists $i \ge 1$ that satisfies $l(Z_{(i)}) = l(Z_{(j)})$ for any $j > i$. We get $a(Z_{(j)}) = (k + 2)n + k + 1$ by (4) and then $a(N_1(Z_{(j)})) = (k + 2)n$ by (5). Therefore $N_1(Z_{(j)})$ is not a variable for any $j > i$ and from (6), we obtain $N_1(Z_{(j)}) = N_1(N_1(Z_{(j-1)})) = \cdots = \underbrace{N_1(\cdots N_1(}_{j-i+1} Z_{(i)}) \cdots)$ for any $j > i$. However, this implies that $Z_{(i)}$ has infinitely many variables and it yields contradiction. ◀

Now, we get the desired result:

▶ **Theorem 16.** *For any $k \ge 0$ and $n > 0$, $(B^k B)^{(k+2)n}$ does not have the ρ-property.*

The key fact which enables us to show the anti-ρ-property of $(B^k B)^{(k+2)n}$ is the existence of the set $T_{k,n} \supset \left\{ ((B^k B)^{(k+2)n})_{(i)} \mid i \ge 1 \right\}$ which satisfies Lemma 14. In a similar way, we can show the anti-ρ-property of a B-term which has such a "good" set. That is,

▶ **Theorem 17.** *Let X be a B-term and T be a set of B-terms. If $\left\{ X_{(i)} \mid i \geq 1 \right\} \subset T$ and $l(X) \geq a(X') + 1$ for any $X' \in T$, then X does not have the ρ-property.*

Here is an example of the B-terms which satisfy the condition in Theorem 17 with some set T. Consider $X = (B^2 B)^2 \circ (BB)^2 \circ B^2 = \langle \star, \ \langle \star, \ \langle \star, \ \langle \star, \ \star, \ \star \rangle, \ \star \rangle, \ \star \rangle \rangle$. We inductively define T' as follows:

1. $\star \in T'$
2. For any $t \in T'$, $\langle \star, \ t, \ \star \rangle \in T'$
3. For any $t_1, t_2 \in T'$, $\langle \star, \ t_1, \ \star, \ \langle \star, \ t_2, \ \star \rangle, \ \star \rangle \in T'$

Then $T = \{ \langle t_1, \ \langle \star, \ t_2, \ \star \rangle \rangle \mid t_1, t_2 \in T' \}$ satisfies the condition in Theorem 17. It can be checked simply by case analysis. Thus

▶ **Theorem 18.** $(B^2 B)^2 \circ (BB)^2 \circ B^2$ *does not have the ρ-property.*

Theorem 17 gives a possible technique to prove the monotonicity with respect to $l(X_{(i)})$, or, the anti-ρ-property of X, for some B-term X. Moreover, we can consider another problem on B-terms: "Give a necessary and sufficient condition to have the monotonicity for B-terms."

5   Concluding remark

We have investigated the ρ-properties of B-terms in particular forms so far. While the B-terms equivalent to $B^n B$ with $n \leq 6$ have the ρ-property, the B-terms $(B^k B)^{(k+2)n}$ with $k \geq 0$ and $n > 0$ and $(B^2 B)^2 \circ (BB)^2 \circ B^2$ do not. In this section, remaining problems related to these results are introduced and possible approaches to illustrate them are discussed.

5.1   Remaining problems

The ρ-property is defined for any combinatory terms (and closed λ-terms). We investigated it only for B-terms as a simple but interesting instance in the present paper. From his observation on repetitive right applications for several B-terms, Nakano [8] has conjectured as follows.

▶ **Conjecture 19.** *A B-term e has the ρ-property if and only if e is a monomial, i.e., e is equivalent to $B^n B$ with $n \geq 0$.*

The "if" part for $n \leq 6$ has been shown by computation and the "only if" part for $(B^k B)^{(k+2)n}$ $(k \geq 0, n > 0)$ and $(B^2 B)^2 \circ (BB)^2 \circ B^2$ has been shown by Theorem 16. This conjecture implies that the ρ-property of B-terms is decidable. We conjecture that the ρ-property of even BCK- and BCI-terms is decidable. The decidability for the ρ-property of S-terms and L-terms can also be considered. Waldmann's work on a rational representation of normalizable S-terms may be helpful to solve it. We expect that none of S-terms have the ρ-property as S itself does not, though. Regarding L-terms, Statman's work [11] may be helpful where equivalence of L-terms is shown decidable up to a congruence relation induced by $L \ e_1 \ e_2 \to e_1 \ (e_2 \ e_2)$. It would be interesting to investigate the ρ-property of L-terms in this setting.

5.2   Possible approaches

The present paper introduces a canonical representation to make equivalence check of B-terms easier. The idea of the representation is based on that we can lift all $\circ$'s (2-argument B) to the outside of B (1-argument B) by equation (B2'). One may consider it the other way around. Using the equation, we can lift all B's (1-argument B) to the outside of $\circ$ (2-argument B).

Then one of the arguments of $\circ$ becomes B. By equation (B3'), we can move all B's right. Thereby we find another canonical representation for B-terms given by

$$e ::= B \mid B\ e \mid e \circ B$$

whose uniqueness could be easily proved in a way similar to Theorem 9.

Waldmann [13] suggests that the ρ-property of $B^n B$ may be checked even without converting B-terms into canonical forms. He simply defines B-terms by

$$e ::= B^k \mid e\ e$$

and regards B^k as a constant which has a rewrite rule $B^k\ e_1\ e_2\ \ldots\ e_{k+2} \to e_1\ (e_2\ \ldots\ e_{k+2})$. He implemented a check program in Haskell to confirm the ρ-property. Even in the restriction on rewriting rules, he found that $(B^0 B)_{(9)} = (B^0 B)_{(13)}$, $(B^1 B)_{(36)} = (B^1 B)_{(56)}$, $(B^2 B)_{(274)} = (B^2 B)_{(310)}$ and $(B^3 B)_{(4267)} = (B^3 B)_{(10063)}$, in which it requires a few more right applications to find the ρ-property than the case of canonical representation. If the ρ-property of $B^n B$ for any $n \geq 0$ is shown under the restricted equivalence given by rewriting rules, then we can conclude the "if" part of Conjecture 19.

Another possible approach is to observe the change of (principal) types by right repetitive application. Although there are many distinct λ-terms of the same type, we can consider a desirable subset of typed λ-terms. As shown by Hirokawa [5], each BCK-term can be characterized by its type, that is, any two λ-terms in $\mathbf{CL}(BCK)$ of the same principal type are identical up to β-equivalence. This approach may require observing unification between types in a clever way.

References

1   Hendrik P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics. Elsevier Science, 1984.

2   Michael Beeler, Ralph W. Gosper, and Richard C. Schroeppel. HAKMEM. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1972.

3   Richard P. Brent. An improved Monte Carlo factorization algorithm. *BIT*, 20(2):176–184, 1980.

4   Haskell B. Curry. Grundlagen der Kombinatorischen Logik (Teil II). *American Journal of Mathematics*, 52(4):789–834, 1930.

5   Sachio Hirokawa. Principal types of BCK-lambda-terms. *Theoretical Computer Science*, 107(2):253–276, Jan 1993.

6   Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

7   Evgeny S. Ljapin. *Semigroups*. Translations of Mathematical Monographs. American Mathematical Society, 1968.

8   Keisuke Nakano. ρ-property of combinators. 29th TRS Meeting in Tokyo, 2008. URL: `http://www.jaist.ac.jp/~hirokawa/trs-meeting/original/29.html`.

9   Chris Okasaki. Flattening combinators: surviving without parentheses. *Journal of Functional Programming*, 13(4):815–822, July 2003.

10   Raymond M. Smullyan. *To Mock a Mockingbird*. Knopf Doubleday Publishing Group, 2012.

11   Rick Statman. The Word Problem for Smullyan's Lark Combinator is Decidable. *Journal of Symbolic Computation*, 7(2):103–112, 1989.

12   Rick Statman. To Type A Mockingbird. Draft paper available from `http://tlca.di.unito.it/PAPER/TypeMock.pdf`, December 2011.

13   Johannes Waldmann. Personal communication, March 2013.

Index-Stratified Types

Rohan Jacob-Rao
Digital Asset, Sydney, Australia
rohanjr@digitalasset.com

Brigitte Pientka
McGill University, Montreal, Canada
bpientka@cs.mcgill.ca

David Thibodeau
McGill University, Montreal, Canada
david.thibodeau@mail.mcgill.ca

──── **Abstract** ────

We present TORES, a core language for encoding metatheoretic proofs. The novel features we introduce are well-founded Mendler-style (co)recursion over indexed data types and a form of recursion over objects in the index language to build new types. The latter, which we call *index-stratified types*, are analogue to the concept of large elimination in dependently typed languages. These features combined allow us to encode sophisticated case studies such as normalization for lambda calculi and normalization by evaluation. We prove the soundness of TORES as a programming and proof language via the key theorems of subject reduction and termination.

1 Introduction

Recursion is a fundamental tool for writing useful programs in functional languages. When viewed from a logical perspective via the Curry-Howard correspondence, well-founded recursion corresponds to inductive reasoning. Dually, well-founded corecursion corresponds to coinductive reasoning. However, concentrating only on well-founded (co)recursive definitions is not sufficient to support the encoding of meta-theoretic proofs. There are two missing ingredients: 1) To express fine-grained properties we often rely on first-order logic which is analogous to *indexed types* in programming languages. 2) Many common notions cannot be directly characterized by well-founded (co)recursive definitions. An example is Girard's notion of reducibility for functions: a term M is reducible at type $A \rightarrow B$ if, for all terms N that are reducible at type A, we have that $M\ N$ is reducible at type B. This definition is

well-founded because it is by structural recursion on the type indices (A and B), so we want
to admit such definitions.

Our contribution in this paper is a core language called TORES that features indexed types
and (co)inductive reasoning via well-founded (co)recursion. The primary forms of types are
indexed (co)recursive types, over which we support reasoning via Mendler-style (co)recursion.
Additionally, TORES features *index-stratified types*, which allow further definitions of types
via well-founded recursion over indices. The main difference between the two forms is that
(co)recursive types are more flexible, allowing (co)induction, while stratified types only
support unfolding based on their indices. The combination of the two features is especially
powerful for formalizing metatheory involving logical relations. This is partly because type
definitions in TORES do not require positivity, a condition used in other systems to ensure
termination and in turn logical consistency. Despite this, we are able to prove termination of
TORES programs using a semantic interpretation of types.

How to justify definitions that are recursively defined on a given index in addition to
well-founded (co)recursive definitions has been explored in proof theory (for example [22, 3]).
While this line of work is more general, it is also more complex and further from standard
programming practice. In dependent type theories, large eliminations achieve the same. Our
approach, grounded in the Curry-Howard isomorphism, provides a complementary perspective
on this problem where we balance expressiveness and ease of programming with a compact
metatheory. We believe this may be an advantage when considering more sophisticated index
languages and reasoning techniques.

The combination of indexed (co)recursive types and stratified types is already used
in the programming and proof environment BELUGA, where the index language is an
extension of the logical framework LF together with first-class contexts and substitutions
[15, 16, 4]. This allows elegant implementations of proofs using logical relations [5, 6] and
normalization by evaluation [4]. TORES can be seen as small kernel into which we elaborate
total BELUGA programs, thereby providing post-hoc justification of viewing BELUGA programs
as (co)inductive proofs.

2 Index Language for Tores

The design of TORES is parametric over an index language. Following Thibodeau et al.
[21] we stay as abstract as possible and state the general conditions the index language
must satisfy. Whenever we require inspection of the particular index language, namely the
structure of stratified types and induction terms, we will draw attention to it.

To illustrate the required structure for a concrete index language, we use natural numbers.
In practice, however, we can consider other index languages such as those of strings, types
[7, 24], or (contextual) LF [16, 4]. It is important to note that, for most of our design, we
accommodate a general index language up to the complexity of Contextual LF. Thus we
treat index types and TORES kinds as dependently typed.

2.1 General Structure

We refer to a term in the index language as an *index term M*, which may have an *index type
U*. In the case of natural numbers, there is a single index type nat, and index terms are built
from 0, suc, and variables u which must be declared in an *index context Δ*.

Index types	U	$:= \mathsf{nat}$	Index contexts	Δ	$:= \cdot \mid \Delta, u{:}U$
Index terms	M	$:= 0 \mid \mathsf{suc}\, M \mid u$	Index substitutions	Θ	$:= \cdot \mid \Theta, M/u$

$\boxed{\vdash \Delta \;\mathsf{ictx}}$ Index context Δ is well-formed $\boxed{\Delta \vdash U \;\mathsf{itype}}$ Index type U is well-kinded

$$\frac{}{\vdash \cdot \;\mathsf{ictx}} \qquad \frac{\vdash \Delta \;\mathsf{ictx} \quad \Delta \vdash U \;\mathsf{itype}}{\vdash \Delta, u{:}U \;\mathsf{ictx}} \qquad\qquad \frac{}{\Delta \vdash \mathsf{nat} \;\mathsf{itype}}$$

$\boxed{\Delta \vdash M : U}$ Index term M has index type U in index context Δ

$$\frac{u{:}U \in \Delta}{\Delta \vdash u : U} \qquad \frac{}{\Delta \vdash 0 : \mathsf{nat}} \qquad \frac{\Delta \vdash M : \mathsf{nat}}{\Delta \vdash \mathsf{suc}\, M : \mathsf{nat}}$$

$\boxed{\Delta \vdash M = N}$ Index term M is equal to N

$$\frac{}{\Delta \vdash M = M}$$

$\boxed{\Delta' \vdash \Theta : \Delta}$ Index substitution Θ maps index variables from Δ to Δ'

$$\frac{}{\Delta' \vdash \cdot : \cdot} \qquad \frac{\Delta' \vdash \Theta : \Delta \quad \Delta' \vdash M : U[\Theta]}{\Delta' \vdash \Theta, M/u : \Delta, u{:}U}$$
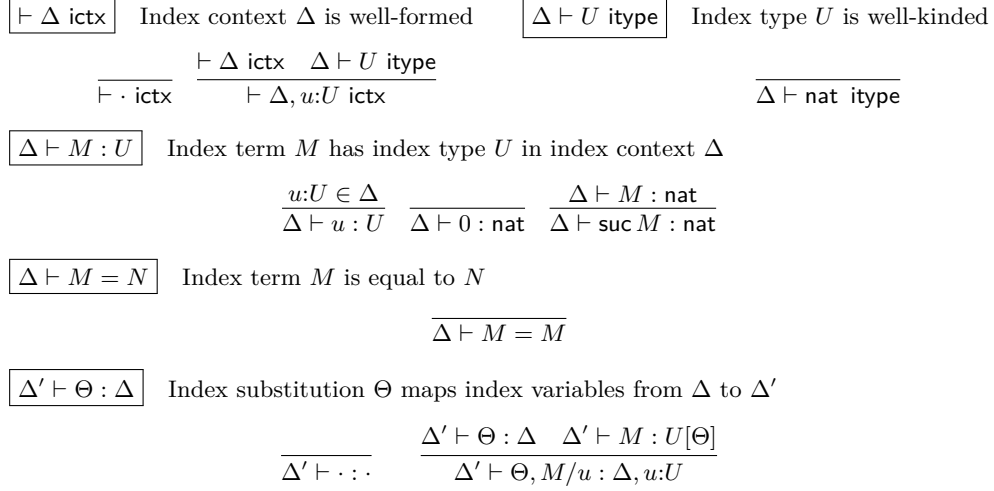
Figure 1 Index language structure.

TORES relies on typing for index terms which we give for natural numbers in Fig. 1. The equality judgment for natural numbers is given simply by reflexivity (syntactic equality). We also give typing for index substitutions, which supply an index term for each index variable in the domain Δ and describe well-formed contexts. These definitions are generic.

We require that both typing and equality of index terms be decidable in order for type checking of TORES programs to be decidable.

▶ **Requirement 1.** *Index type checking is decidable.*

▶ **Requirement 2.** *Index equality is decidable.*

We can lift the kinding, typing, equality and matching rules to *spines* of index terms and types generically. We write $\cdot$ and $(\cdot)$ for the empty spines of terms and types respectively. If M_0 is an index term and $\overrightarrow{M}$ is a spine, then $M_0, \overrightarrow{M}$ is a spine. Similarly if $u_0{:}U_0$ is an index type assignment and $(\overrightarrow{u{:}U})$ is a type spine, then $(u_0{:}U_0, \overrightarrow{u{:}U})$ is a type spine. Spines are convenient for setting up the types and terms of TORES. Unlike index substitutions Θ which are built from right to left, spines are built from left to right.

Throughout our development we use both a single index substitution operation $M[N/u]$ and a simultaneous substitution operation $M[\Theta]$. For composition of simultaneous substitutions we write $\Theta_1[\Theta_2]$.

▶ **Requirement 3** (Index substitution principles)**.**

3.1. *If $\Delta_1, u{:}U', \Delta_2 \vdash M : U$ and $\Delta_1 \vdash N : U'$ then $\Delta_1, \Delta_2[N/u] \vdash M[N/u] : U[N/u]$.*
3.2. *If $\Delta' \vdash \Theta : \Delta$ and $\Delta \vdash M : U$ then $\Delta' \vdash M[\Theta] : U[\Theta]$.*
3.3. *If $\Delta' \vdash \Theta : \Delta$ and $\Delta \vdash M = N$ then $\Delta' \vdash M[\Theta] = N[\Theta]$.*
3.4. *If $\Delta \vdash M : U$ and $\Delta_1 \vdash \Theta_1 : \Delta$ and $\Delta_2 \vdash \Theta_2 : \Delta_1$ then $M[\Theta_1][\Theta_2] = M[\Theta_1[\Theta_2]]$.*

2.2 Unification and Matching

Type checking of TORES relies on a unification procedure to generate a *most general unifier (MGU)*. A *unifier* for index terms M and N in a context Δ is a substitution Θ which transforms

M and N into syntactically equal terms in another context Δ'. That is, $\Delta' \vdash \Theta : \Delta$ and $\Delta' \vdash M[\Theta] = N[\Theta]$. Θ is "most general" if it does not make more commitments to variables than absolutely necessary. A unifying substitution Θ only makes sense together with its range Δ', so we usually write them as a pair $(\Delta' \mid \Theta)$. In general, there may be more than one MGU for a particular unification problem, or none at all. However, we require here that each problem has at most one MGU up to α-equivalence. We write the generation of an MGU using the judgment $\Delta \vdash M \doteq N \searrow P$, where P is either the MGU $(\Delta' \mid \Theta)$ if it exists or $\#$ representing that unification failed. To illustrate, we show the unification rules for natural numbers. We write id_i for the identity substitution that maps index variables from Δ_i to themselves.

$$\boxed{\Delta \vdash M \doteq N \searrow P} \quad \text{Index terms } M \text{ and } N \text{ have MGU } P$$

$$\frac{}{\Delta \vdash 0 \doteq 0 \searrow (\Delta \mid \mathsf{id})} \quad \frac{\Delta \vdash M \doteq N \searrow P}{\Delta \vdash \mathsf{suc}\, M \doteq \mathsf{suc}\, N \searrow P} \quad \frac{}{\Delta \vdash u \doteq u \searrow (\Delta \mid \mathsf{id})}$$

$$\frac{u \notin \mathrm{FV}(M) \quad \Delta = \Delta_1, u{:}U, \Delta_2 \quad \Delta' = \Delta_1, \Delta_2[M/u]}{\Delta \vdash u \doteq M \searrow (\Delta' \mid \mathsf{id}_1, M/u, \mathsf{id}_2)} \qquad (\text{same for } M \doteq u)$$

$$\frac{}{\Delta \vdash 0 \doteq \mathsf{suc}\, M \searrow \#} \quad \frac{}{\Delta \vdash \mathsf{suc}\, M \doteq 0 \searrow \#} \quad \frac{u \in \mathrm{FV}(M) \quad M \neq u}{\Delta \vdash u \doteq M \searrow \#} \quad (\text{same for } M \doteq u)$$

▶ **Requirement 4** (Decidable unification). *Given index terms M and N in a context Δ, the judgment $\Delta \vdash M \doteq N \searrow P$ is decidable. Either P is $(\Delta' \mid \Theta)$, the unique MGU up to α-equivalence, or P is $\#$ and there is no unifier.*

Finally, our operational semantics relies on index *matching*. This is an asymmetric form of unification: given terms M in Δ and N in Δ', matching identifies a substitution Θ such that $\Delta' \vdash M[\Theta] = N$. We describe it using the judgment $\Delta \vdash M \doteq N \searrow (\Delta' \mid \Theta)$. The notion of matching also lifts to the level of index substitutions. We omit the full specifications here and instead state the required properties.

▶ **Requirement 5** (Soundness of index matching).

5.1. *If $\Delta \vdash M : U$ and $\Delta \vdash M \doteq N \searrow (\Delta' \mid \Theta)$ then $\Delta' \vdash \Theta : \Delta$ and $\Delta' \vdash M[\Theta] = N$.*
5.2. *If $\Delta_1 \vdash \Theta_1 : \Delta$ and $\Delta_1 \vdash \Theta_1 \doteq \Theta_2 \searrow (\Delta_2 \mid \Theta)$ then $\Delta_2 \vdash \Theta : \Delta_1$ and $\Delta_2 \vdash \Theta_1[\Theta] = \Theta_2$.*

▶ **Requirement 6** (Completeness of index matching). *Suppose $\vdash \theta : \Delta$ and $\vdash M[\theta] = N[\theta]$ and $\Delta \vdash M \doteq N \searrow (\Delta' \mid \Theta)$. Then $\Delta' \vdash \Theta \doteq \theta \searrow (\cdot \mid \theta')$.*

3   Specification of Tores

We now describe Tores, a programming language designed to express (co)inductive proofs and programs using Mendler-style (co)recursion. It also features *index-stratified* types, which allow definitions of types via well-founded recursion over indices.

3.1   Types and Kinds

Besides unit, products and sums, Tores includes a nonstandard function type $\overrightarrow{(u{:}U)}; T_1 \to T_2$, which combines a dependent function type and a simple function type. It binds a number of index variables $\overrightarrow{u{:}U}$ which may appear in both T_1 and T_2. If the spine of type declarations

is empty then $(\cdot); T_1 \to T_2$ degenerates to the simple function space. We can also quantify existentially over an index using the type $\Sigma u{:}U.\,T$, and have a type for index equality $M = N$. These two types are useful for expressing equality constraints on indices. We model (co)recursive and stratified types as type constructors of kind $\overrightarrow{\Pi u{:}U}.\,*$. These introduce type variables X, which we track in the type variable context Ξ. There is no positivity condition on recursive types, as the typing rules for Mendler-recursion enforce termination without it.

A stratified type is defined by primitive recursion on an index term. For the index type nat, the two branches correspond to the two constructors 0 and suc. Intuitively, $T_{\text{Rec}}\, 0$ will behave like T_0 and $T_{\text{Rec}}\,(\text{suc}\, M)$ will behave like $T_s[M/u, T_{\text{Rec}}\, M/X]$. For richer index languages such as Contextual LF we can generate an appropriate recursion scheme following Pientka and Abel [17].

Kinds	K	$::= * \mid \Pi u{:}U.\,K$
Types	T	$::= 1 \mid T_1 \times T_2 \mid T_1 + T_2 \mid (\overrightarrow{u{:}U}); T_1 \to T_2 \mid \Sigma u{:}U.\,T \mid M = N$
		$\mid T\,M \mid \Lambda u.\,T \mid X \mid \mu X{:}K.\,T \mid \nu X{:}K.\,T \mid T_{\text{Rec}}$
Stratified Types	T_{Rec}	$::= \text{Rec}_K(0 \mapsto T_0 \mid \text{suc}\, u,\, X \mapsto T_s)$
Index Contexts	Δ	$::= \cdot \mid \Delta, u{:}U$
Type Var. Contexts	Ξ	$::= \cdot \mid \Xi, X{:}K$
Typing Contexts	Γ	$::= \cdot \mid \Gamma, x{:}T$

▶ **Example 1.** We illustrate indexed recursive types and stratified types using vectors, i.e. lists indexed by their length, with elements of type A. Vectors are of kind $\Pi n{:}\,\text{nat}\,.*$. We omit the kind annotation for better readability in the subsequent type definitions. One way to define vectors is with an indexed recursive type, an explicit equality and an existential type: $\text{Vec}_\mu \equiv \mu V.\,\Lambda n.\,n = 0 + \Sigma m{:}\,\text{nat}\,.\,n = \text{suc}\, m \times (A \times V\, m)$.

Alternatively, they can be defined as a stratified type: $\text{Vec}_S \equiv \text{Rec}\,(0 \mapsto 1 \mid \text{suc}\, m,\, V \mapsto A \times V)$. In this case equality reasoning is implicit. While we have a choice how to define vectors, some types are only possible to encode using one form or the other.

▶ **Example 2.** A type that must be stratified is the encoding of reducibility for simply typed lambda terms. This example is explored in detail by Cave and Pientka [5]; our work gives it theoretical justification.

Here the index objects are the simple types, unit and arr $A\, B$ of index type tp, as well as lambda terms (), lam $x.M$ and app $M\, N$ of index type tm. We can define reducibility as a stratified type of kind $\Pi a{:}\text{tp}.\Pi m{:}\text{tm}.*$. This relies on an indexed recursive type Halt (omitted here) that describes when a term m steps to a value.

$$\text{Red} \equiv \text{Rec}\,(\; \text{unit} \qquad \mapsto \Lambda m.\text{Halt}\, m$$
$$\mid \text{arr}\, a\, b, R_a, R_b \mapsto \Lambda m.\text{Halt}\, m \times (n{:}\text{tm}); R_a\, n \to R_b\,(\text{app}\, m\, n)\;)$$

▶ **Example 3.** To illustrate a corecursive type, we define an indexed stream of bits following Thibodeau et al. [21]. The index here guarantees that we are reading exactly m bits. Once $m = 0$, we read a new message consisting of the length of the message n together with a stream indexed by n. In contrast to the recursive type definition for vectors, here the equality constraints guard the observations we can make about a stream.

$$\text{Stream} \equiv \nu \text{Str}.\,\Lambda m.\quad (\cdot);\, m = 0 \quad\to \Sigma n{:}\,\text{nat}\,.\,\text{Str}\, n$$
$$\times\, (n{:}\text{nat});\, m = \text{suc}\, n \to \text{Str}\, n$$
$$\times\, (n{:}\text{nat});\, m = \text{suc}\, n \to \text{Bit}$$

3.2   Terms

Tores contains many common constructs found in functional programming languages, such as unit, pairs and case expressions. We focus on the less standard constructs: indexed functions, equality witnesses, well-founded recursion and index induction.

$$
\begin{aligned}
\text{Terms } t, s ::={} & x \mid \langle \rangle \mid \lambda \vec{u}, x.\, t \mid t\, \vec{M}\, s \mid \langle t_1, t_2 \rangle \mid \texttt{split}\, s\, \texttt{as}\, \langle x_1, x_2 \rangle\, \texttt{in}\, t \\
& \mid \texttt{in}_i\, t \mid (\texttt{case}\, t\, \texttt{of}\, \texttt{in}_1\, x_1 \mapsto t_1 \mid \texttt{in}_2\, x_2 \mapsto t_2) \\
& \mid \texttt{pack}\, (M, t) \mid \texttt{unpack}\, t\, \texttt{as}\, (u, x)\, \texttt{in}\, s \\
& \mid \texttt{refl} \mid \texttt{eq}\, s\, \texttt{with}\, (\Delta.\Theta \mapsto t) \mid \texttt{eq\_abort}\, s \\
& \mid \texttt{in}_\mu\, t \mid \texttt{rec}\, f.\, t \mid \texttt{corec}\, f.\, t \mid \texttt{out}_\nu\, t \mid \texttt{in}_l\, t \mid \texttt{out}_l\, t \mid \texttt{ind}\, t_0\, (u,\, f.\, t_s) \mid t{:}T
\end{aligned}
$$

Since we combine the dependent and simple function types in $(\overrightarrow{u{:}U});\, T_1 \to T_2$, we similarly combine abstraction over index variables $\vec{u}$ and a term variable x in our function term $\lambda \vec{u}, x.\, t$. The corresponding application form is $t\, \vec{M}\, s$. The term t of function type $(\overrightarrow{u{:}U});\, T_1 \to T_2$ receives first a spine $\vec{M}$ of index objects followed by a term s. Each equality type $M = N$ has at most one inhabitant $\texttt{refl}$ witnessing the equality. There are two elimination forms for equality: the term $\texttt{eq}\, s\, \texttt{with}\, (\Delta.\Theta \mapsto t)$ uses an equality proof s for $M = N$ together with a unifier Θ to refine the body t in a new index context Δ. It may also be the case that the equality witness s is false, in which case we have reached a contradiction and abort using the term $\texttt{eq\_abort}\, s$. Both forms are necessary to make use of equality constraints that arise from indexed type definitions and to show that some cases are impossible.

Recursive types are introduced by the "fold" syntax $\texttt{in}_\mu$, and stratified types are introduced by $\texttt{in}_l$ terms. Here l ranges over constructors in the index language such as 0 and suc. The important difference is how we eliminate recursive and stratified types. We can analyze data defined by a recursive type using Mendler-style recursion $\texttt{rec}\, f.\, t$. This gives a powerful means of recursion while still ensuring termination. Stratified types can only be unfolded using $\texttt{out}_l$ according to the index. To take full advantage of stratified types, we also allow programmers to use well-founded recursion over index objects, writing $\texttt{ind}\, t_0\, (u,\, f.\, t_s)$. Intuitively, if the index object is 0, then we pick the first branch and execute t_0; if the index object is suc M then we pick the second branch instantiating u with M and allowing recursive calls f inside t_s. While this induction principle is specific to natural numbers, it can also be derived for other index domains, in particular contextual LF (see Pientka and Abel [17]).

▶ **Example 4.** Recall that vectors can be defined using the indexed recursive type Vec_μ or the stratified type Vec_S. Which definition we choose impacts how we write programs that analyze vectors. We show the difference using a recursive function that copies a vector.

$$
\begin{aligned}
copy : (n{:}\,&\mathsf{nat});\, \mathsf{Vec}_\mu\, n \to \mathsf{Vec}_\mu\, n \equiv \texttt{rec}\, f.\, \lambda n, v.\, \texttt{case}\, v\, \texttt{of} \\
& \mid \texttt{in}_1\, z \quad \mapsto \quad \texttt{in}_\mu\, (\texttt{in}_1\, z) \\
& \mid \texttt{in}_2\, s \quad \mapsto \quad \texttt{unpack}\, s\, \texttt{as}\, (m, p)\, \texttt{in} \\
& \qquad\qquad\qquad \texttt{split}\, p\, \texttt{as}\, \langle e, p' \rangle\, \texttt{in} \\
& \qquad\qquad\qquad \texttt{split}\, p'\, \texttt{as}\, \langle h, t \rangle\, \texttt{in} \\
& \qquad\qquad\qquad \texttt{in}_\mu\, (\texttt{in}_2\, (\texttt{pack}\, (m, \langle e, \langle h, f\, m\, t \rangle \rangle)))
\end{aligned}
$$

To analyze the recursively defined vector, we use recursion and case analysis of the input vector to reconstruct the output vector. If we receive a non-empty list, we take it apart and expose the equality proofs, before reassembling the list. The recursion is valid according to the Mendler typing rule since the recursive call to f is made on the tail of the input vector.

To contrast we show the program using induction on natural numbers and unfolding the stratified type definition of Vec_S. Note that the first argument is the natural number index n paired with a unit term argument, since index abstraction is always combined with term

abstraction. The program analyzes n and in the suc case unfolds the input vector before reconstructing it using the result of the recursive call. In this version of *copy* the equality constraints are handled silently by the type checker.

$$
\begin{aligned}
&copy : (n\!:\mathsf{nat});\ 1 \to (\cdot);\ \mathsf{Vec}_S\, n \to \mathsf{Vec}_S\, n \equiv\\
&\quad \mathtt{ind}\,(\ 0 \qquad\qquad \mapsto \quad \lambda\,v.\,\mathtt{in}_0\,\langle\rangle\\
&\qquad\quad |\ \mathsf{suc}\,m,\, f_m \quad \mapsto \quad \lambda\,v.\,\mathtt{split}\,(\mathsf{out}_{\mathsf{suc}}\,v)\,\mathtt{as}\,\langle h,t\rangle\,\mathtt{in}\quad \mathtt{in}_{\mathsf{suc}}\,\langle h, f_m\,t\rangle)
\end{aligned}
$$

▶ **Example 5.** Note that TORES does not have an explicit notion of falsehood. This is because it is definable using existing constructs: we can define the empty type as a recursive type $\bot \equiv \mu X\!:\ast.\,X$, and a contradiction term $\mathtt{abort} \equiv \mathtt{rec}\,f.\,f : \bot \to C$, for any type C. Our termination result with the logical relation in Section 4.3 shows that the $\bot$ type contains no values and hence no closed terms, which implies logical consistency of TORES (not all propositions can be proven).

3.3 Typing Rules

We define a bidirectional type system in Fig. 2. We focus here on equality, recursive and stratified types.

The introduction for an index equality type is simply $\mathtt{refl}$, which is checked via equality in the index domain. Both equality elimination forms rely on unification in the index domain (see Section 2.2). Specifically, the $\mathtt{eq\_abort}\,s$ term checks against any type because the unification must fail, establishing a contradiction. For the term $\mathtt{eq}\,s\,\mathtt{with}\,(\Delta'.\Theta \mapsto t)$, unification must result in the MGU which by Req. 4 is α-equivalent to the supplied unifier $(\Delta' \mid \Theta)$. We then check the body t using the new index context Δ' and Θ applied to the contexts Ξ and Γ and the goal type T.

This treatment of equality elimination is similar to the use of refinement substitutions for dependent pattern matching [18, 4], and is inspired by equality elimination in proof theory [23, 12, 19]. In the latter line of work, type checking involves trying all unifiers from a *complete set of unifiers* (which may be infinite!), instead of a single most general unifier. We believe our requirement for a unique MGU is a practical choice for type checking.

Indexed recursive and stratified types are both introduced by injections ($\mathtt{in}_\mu$ and $\mathtt{in}_l$), though their elimination forms are different. Stratified types are eliminated (unfolded) in reverse to the corresponding fold rules. For recursive types on the other hand, the naive unfold rules lead to nontermination, so we use a Mendler-style recursion form $\mathtt{rec}\,f.\,t$, generalizing the original formulation [13] to an indexed type system. The idea is to constrain the type of the function variable f so that it can only be applied to structurally smaller data. This is achieved by declaring f of type $(\overrightarrow{u\!:\!U});\ X\,\vec{u} \to T$ in the premise of the rule. Here X represents types exactly one constructor smaller than the recursive type, so the use of f is guaranteed to be well-founded.

▶ **Theorem 6.** *Type checking of terms is decidable.*

Proof. Since the typing rules are syntax directed, it is straight-forward to extract a type checking algorithm. Note that the algorithm relies on decidability of judgments in the index language, namely index type checking (Req. 1), equality (Req. 2) and unification (Req. 4). ◀

$\boxed{\Delta;\Xi;\Gamma \vdash t \Leftarrow T}$ Term t checks against input type T

$$\frac{\Delta;\Xi;\Gamma \vdash t_1 \Leftarrow T_1 \quad \Delta;\Xi;\Gamma \vdash t_2 \Leftarrow T_2}{\Delta;\Xi;\Gamma \vdash \langle t_1, t_2 \rangle \Leftarrow T_1 \times T_2} \quad \frac{\Delta;\Xi;\Gamma \vdash p \Rightarrow T_1 \times T_2 \quad \Delta;\Xi;\Gamma, x_1{:}T_1, x_2{:}T_2 \vdash t \Leftarrow T}{\Delta;\Xi;\Gamma \vdash \mathtt{split}\, p \,\mathtt{as}\, \langle x_1, x_2 \rangle \,\mathtt{in}\, t \Leftarrow T}$$

$$\frac{\Delta;\Xi;\Gamma \vdash t \Leftarrow T_i}{\Delta;\Xi;\Gamma \vdash \mathtt{in}_i\, t \Leftarrow T_1 + T_2} \quad \frac{\Delta;\Xi;\Gamma \vdash t \Rightarrow T_1 + T_2 \quad \Delta;\Xi;\Gamma, x_1{:}T_1 \vdash t_1 \Leftarrow S \quad \Delta;\Xi;\Gamma, x_2{:}T_2 \vdash t_2 \Leftarrow S}{\Delta;\Xi;\Gamma \vdash (\mathtt{case}\, t \,\mathtt{of}\, \mathtt{in}_1\, x_1 \mapsto t_1 \mid \mathtt{in}_2\, x_2 \mapsto t_2) \Leftarrow S}$$

$$\frac{\Delta \vdash M : U \quad \Delta;\Xi;\Gamma \vdash t \Leftarrow T[M/u]}{\Delta;\Xi;\Gamma \vdash \mathtt{pack}\,(M, t) \Leftarrow \Sigma u{:}U.\, T} \quad \frac{\Delta;\Xi;\Gamma \vdash t \Rightarrow \Sigma u{:}U.\, T \quad \Delta, u{:}U;\Xi;\Gamma, x{:}T \vdash s \Leftarrow S}{\Delta;\Xi;\Gamma \vdash \mathtt{unpack}\, t \,\mathtt{as}\, (u, x) \,\mathtt{in}\, s \Leftarrow S}$$

$$\frac{\Delta \vdash M = N}{\Delta;\Xi;\Gamma \vdash \mathtt{refl} \Leftarrow M = N} \quad \frac{\Delta;\Xi;\Gamma \vdash s \Rightarrow M = N \quad \Delta \vdash M \doteq N \searrow \#}{\Delta;\Xi;\Gamma \vdash \mathtt{eq\_abort}\, s \; \Leftarrow T}$$

$$\frac{\Delta;\Xi;\Gamma \vdash s \Rightarrow M = N \quad \Delta \vdash M \doteq N \searrow (\Delta' \mid \Theta) \quad \Delta';\Xi[\Theta];\Gamma[\Theta] \vdash t \Leftarrow T[\Theta]}{\Delta;\Xi;\Gamma \vdash \mathtt{eq}\, s \,\mathtt{with}\, (\Delta'.\Theta \mapsto t) \Leftarrow T}$$

$$\frac{\Delta;\Xi;\Gamma \vdash t \Leftarrow T[\overrightarrow{M/u}; \mu X{:}K.\, \Lambda\, \vec{u}.\, T/X]}{\Delta;\Xi;\Gamma \vdash \mathtt{in}_\mu\, t \Leftarrow (\mu X{:}K.\, \Lambda\, \vec{u}.\, T)\, \vec{M}} \quad \frac{\Delta;\Xi, X{:}K;\Gamma, f{:}((\overrightarrow{u{:}U});\, X\vec{u} \to T) \vdash t \Leftarrow (\overrightarrow{u{:}U});\, S[\overrightarrow{u/v}] \to T}{\Delta;\Xi;\Gamma \vdash \mathtt{rec}\, f.\, t \Leftarrow (\overrightarrow{u{:}U});\, (\mu X{:}K.\, \Lambda\, \overrightarrow{v}.\, S)\, \vec{u} \to T}$$

$$\frac{\Delta;\Xi, X{:}K;\Gamma, f{:}((\overrightarrow{u{:}U});\, S \to X\vec{u}) \vdash t \Leftarrow (\overrightarrow{u{:}U});\, S \to T[\overrightarrow{u/v}]}{\Delta;\Xi;\Gamma \vdash \mathtt{corec}\, f.\, t \Leftarrow (\overrightarrow{u{:}U});\, S \to (\nu X{:}K.\, \Lambda\, \overrightarrow{v}.\, T)\, \vec{u}} \quad \frac{\Delta, \overrightarrow{u{:}U};\Xi;\Gamma, x{:}S \vdash t \Leftarrow T}{\Delta;\Xi;\Gamma \vdash \lambda\, \vec{u}, x.\, t \Leftarrow (\overrightarrow{u{:}U});\, S \to T}$$

$$\frac{\Delta;\Xi;\Gamma \vdash t_0 \Leftarrow T[0/u] \quad \Delta, u{:}\mathtt{nat};\Xi;\Gamma, f{:}T \vdash t_s \Leftarrow T[\mathtt{suc}\, u/u]}{\Delta;\Xi;\Gamma \vdash \mathtt{ind}\, t_0\, (u,\, f.\, t_s) \Leftarrow (u{:}\mathtt{nat});\, 1 \to T} \quad \frac{}{\Delta;\Xi;\Gamma \vdash \langle\rangle \Leftarrow 1}$$

$$\frac{\Delta;\Xi;\Gamma \vdash t \Leftarrow T_0\, \vec{M}}{\Delta;\Xi;\Gamma \vdash \mathtt{in}_0\, t \Leftarrow T_{\mathtt{Rec}}\, 0\, \vec{M}} \quad \frac{\Delta;\Xi;\Gamma \vdash t \Leftarrow T_s[N/u; (T_{\mathtt{Rec}}\, N)/X]\, \vec{M}}{\Delta;\Xi;\Gamma \vdash \mathtt{in}_{\mathtt{suc}}\, t \Leftarrow T_{\mathtt{Rec}}\, (\mathtt{suc}\, N)\, \vec{M}} \quad \frac{\Delta;\Xi;\Gamma \vdash t \Rightarrow T}{\Delta;\Xi;\Gamma \vdash t \Leftarrow T}$$

$\boxed{\Delta;\Xi;\Gamma \vdash t \Rightarrow T}$ Term t synthesizes output type T

$$\frac{x{:}T \in \Gamma}{\Delta;\Xi;\Gamma \vdash x \Rightarrow T} \quad \frac{\Delta;\Xi;\Gamma \vdash t \Rightarrow (\overrightarrow{u{:}U});\, S \to T \quad \Delta \vdash \vec{M} : (\overrightarrow{u{:}U}) \quad \Delta;\Xi;\Gamma \vdash s \Leftarrow S[\overrightarrow{M/u}]}{\Delta;\Xi;\Gamma \vdash t\, \vec{M}\, s \Rightarrow T[\overrightarrow{M/u}]}$$

$$\frac{\Delta;\Xi;\Gamma \vdash t \Rightarrow T_{\mathtt{Rec}}\, 0\, \vec{M}}{\Delta;\Xi;\Gamma \vdash \mathtt{out}_0\, t \Rightarrow T_0\, \vec{M}} \quad \frac{\Delta;\Xi;\Gamma \vdash t \Rightarrow T_{\mathtt{Rec}}\, (\mathtt{suc}\, N)\, \vec{M}}{\Delta;\Xi;\Gamma \vdash \mathtt{out}_{\mathtt{suc}}\, t \Rightarrow T_s[N/u; (T_{\mathtt{Rec}}\, N)/X]\, \vec{M}} \quad \frac{\Delta;\Xi;\Gamma \vdash t \Leftarrow T}{\Delta;\Xi;\Gamma \vdash t{:}T \Rightarrow T}$$

$$\frac{\Delta;\Xi;\Gamma \vdash t \Rightarrow (\nu X{:}K.\, \Lambda\, \vec{u}.\, T)\, \vec{M}}{\Delta;\Xi;\Gamma \vdash \mathtt{out}_\nu\, t \Rightarrow T[\overrightarrow{M/u}; \nu X{:}K.\, \Lambda\, \vec{u}.\, T/X]}$$

🟨 **Figure 2** Typing rules for TORES.

3.4 Operational Semantics

We define a big-step operational semantics using environments, which provide closed values for the free variables that may occur in a term.

Term environments	σ	$:= \cdot \mid \sigma, v/x$
Function values	g	$:= \lambda\, \vec{u}, x.\, t \mid \mathtt{rec}\, f.\, t \mid \mathtt{corec}\, f.\, t \mid \mathtt{ind}\, t_0\, (u,\, f.\, t_s)$
Closures	c	$:= (g)[\theta; \sigma] \mid (\mathtt{corec}\, f.\, t)[\theta; \sigma] \cdot \vec{N}\, v$
Values	v	$:= c \mid \langle\rangle \mid \langle v_1, v_2 \rangle \mid \mathtt{in}_i\, v \mid \mathtt{pack}\,(M, v) \mid \mathtt{refl} \mid \mathtt{in}_\mu\, v \mid \mathtt{in}_l\, v$

Values consist of unit, pairs, injections, reflexivity, and closures. Typing for values and environments, which is used to state the subject reduction theorem, are given in the appendix.

The main evaluation judgment, $t[\theta; \sigma] \Downarrow v$, describes the evaluation of a term t under environments $\theta; \sigma$ to a value v. Here, t stands for a term in an index context Δ and term

$$\boxed{t[\theta;\sigma] \Downarrow v} \quad \text{Term } t \text{ under environments } \theta \text{ and } \sigma \text{ evaluates to } v$$

$$\frac{\sigma(x) = v}{x[\theta;\sigma] \Downarrow v} \qquad \overline{\langle\rangle[\theta;\sigma] \Downarrow \langle\rangle} \qquad \frac{t_1[\theta;\sigma] \Downarrow v_1 \quad t_2[\theta;\sigma] \Downarrow v_2}{\langle t_1, t_2\rangle[\theta;\sigma] \Downarrow \langle v_1, v_2\rangle} \qquad \frac{t[\theta;\sigma] \Downarrow \langle v_1, v_2\rangle \quad s[\theta;\sigma, v_1/x_1, v_2/x_2] \Downarrow v}{(\mathtt{split}\, t \,\mathtt{as}\, \langle x_1, x_2\rangle \,\mathtt{in}\, s)[\theta;\sigma] \Downarrow v}$$

$$\frac{t[\theta;\sigma] \Downarrow v}{(\mathtt{in}_i\, t)[\theta;\sigma] \Downarrow \mathtt{in}_i\, v} \qquad \frac{t[\theta;\sigma] \Downarrow \mathtt{in}_i\, v' \quad t_i[\theta;\sigma, v'/x_i] \Downarrow v}{(\mathtt{case}\, t \,\mathtt{of}\, \mathtt{in}_1\, x_1 \mapsto t_1 \mid \mathtt{in}_2\, x_2 \mapsto t_2)[\theta;\sigma] \Downarrow v} \qquad \frac{t[\theta;\sigma] \Downarrow v}{(t{:}T)[\theta;\sigma] \Downarrow v}$$

$$\frac{t[\theta;\sigma] \Downarrow v}{(\mathtt{pack}\,(M, t))[\theta;\sigma] \Downarrow \mathtt{pack}\,(M[\theta], v)} \qquad \frac{t[\theta;\sigma] \Downarrow \mathtt{pack}\,(N, v') \quad s[\theta, N/u; \sigma, v'/x] \Downarrow v}{(\mathtt{unpack}\, t \,\mathtt{as}\,(u, x) \,\mathtt{in}\, s)[\theta;\sigma] \Downarrow v}$$

$$\overline{\mathtt{refl}[\theta;\sigma] \Downarrow \mathtt{refl}} \qquad \frac{s[\theta;\sigma] \Downarrow \mathtt{refl} \quad \Delta \vdash \Theta \doteq \theta \searrow (\cdot \mid \theta') \quad t[\theta';\sigma] \Downarrow v}{(\mathtt{eq}\, s \,\mathtt{with}\,(\Delta.\Theta \mapsto t))[\theta;\sigma] \Downarrow v} \qquad \frac{t[\theta;\sigma] \Downarrow v}{(\mathtt{in}_l\, t)[\theta;\sigma] \Downarrow \mathtt{in}_l\, v}$$

$$\overline{(\lambda\, \vec{u}, x.\, t)[\theta;\sigma] \Downarrow (\lambda\, \vec{u}, x.\, t)[\theta;\sigma]} \qquad \overline{(\mathtt{rec}\, f.\, t)[\theta;\sigma] \Downarrow (\mathtt{rec}\, f.\, t)[\theta;\sigma]} \qquad \frac{t[\theta;\sigma] \Downarrow \mathtt{in}_l\, v}{(\mathtt{out}_l\, t)[\theta;\sigma] \Downarrow v}$$

$$\overline{(\mathtt{corec}\, f.\, t)[\theta;\sigma] \Downarrow (\mathtt{corec}\, f.\, t)[\theta;\sigma]} \qquad \frac{t[\theta;\sigma] \Downarrow c \quad c \cdot_{\mathtt{out}_\nu} \Downarrow w}{(\mathtt{out}_\nu\, t)[\theta;\sigma] \Downarrow w}$$

$$\overline{(\mathtt{ind}\, t_0\,(u,\, f.\, t_s))[\theta;\sigma] \Downarrow (\mathtt{ind}\, t_0\,(u,\, f.\, t_s))[\theta;\sigma]} \qquad \frac{t[\theta;\sigma] \Downarrow c \quad s[\theta;\sigma] \Downarrow v \quad c \cdot \overrightarrow{M[\theta]}\, v \Downarrow w}{(t\, \vec{M}\, s)[\theta;\sigma] \Downarrow w}$$

$$\boxed{c \cdot \vec{N}\, v \Downarrow w} \quad \text{Closure } c \text{ applied to values } \vec{N} \text{ and } v \text{ evaluates to } w$$

$$\frac{t[\theta, \overrightarrow{N/u}; \sigma, v/x] \Downarrow w}{(\lambda\, \vec{u}, x.\, t)[\theta;\sigma] \cdot \vec{N}\, v \Downarrow w} \qquad \frac{t[\theta;\sigma, (\mathtt{rec}\, f.\, t)[\theta;\sigma]/f] \Downarrow c \quad c \cdot \vec{N}\, v \Downarrow w}{(\mathtt{rec}\, f.\, t)[\theta;\sigma] \cdot \vec{N}\,(\mathtt{in}_\mu\, v) \Downarrow w}$$

$$\overline{(\mathtt{corec}\, f.\, t)[\theta;\sigma] \cdot \vec{N}\, v \Downarrow (\mathtt{corec}\, f.\, t)[\theta;\sigma] \cdot \vec{N}\, v}$$

$$\frac{t_0[\theta;\sigma] \Downarrow w}{(\mathtt{ind}\, t_0\,(u,\, f.\, t_s))[\theta;\sigma] \cdot 0\, \langle\rangle \Downarrow w} \qquad \frac{(\mathtt{ind}\, t_0\,(u,\, f.\, t_s))[\theta;\sigma] \cdot N\, \langle\rangle \Downarrow v \quad t_s[\theta, N/u; \sigma, v/f] \Downarrow w}{(\mathtt{ind}\, t_0\,(u,\, f.\, t_s))[\theta;\sigma] \cdot (\mathtt{suc}\, N)\, \langle\rangle \Downarrow w}$$

$$\boxed{c \cdot_{\mathtt{out}_\nu} \Downarrow w} \quad \text{Closure } c \text{ applied to observation } \mathtt{out}_\nu \text{ evaluates to } w$$

$$\frac{t[\theta;\sigma, (\mathtt{corec}\, f.\, t)[\theta;\sigma]/f] \Downarrow c \quad c \cdot \vec{N}\, v \Downarrow w}{((\mathtt{corec}\, f.\, t)[\theta;\sigma] \cdot \vec{N}\, v) \cdot_{\mathtt{out}_\nu} \Downarrow w}$$

Figure 3 Big-step evaluation rules.

variable context Γ. The index environment θ provides closed index objects for all the index variables in Δ, while σ provides closed values for all the variables declared in Γ, i.e. $\vdash \theta : \Delta$ and $\sigma : \Gamma[\theta]$. For convenience, we factor out the application of a closure c to values $\vec{N}$ and v resulting in a value w, using a second judgment written $c \cdot \vec{N}\, v \Downarrow w$. This allows us to treat application of functions (lambdas, recursion and induction) uniformly. Similarly, we factor out the application of $\mathtt{out}_\nu$ to a closure c in an additional judgment written $c \cdot_{\mathtt{out}_\nu} \Downarrow w$. This simplifies the type interpretation used to prove termination.

We only explain the evaluation rule for equality elimination $\mathtt{eq}\, s \,\mathtt{with}\,(\Delta.\Theta \mapsto t)$. We first evaluate the equality witness s under environments $\theta;\sigma$ to the value $\mathtt{refl}$. This ensures that θ respects the index equality $M = N$ witnessed by s. From type checking we know that $\Delta \vdash M[\Theta] = N[\Theta]$: the key is how we extend Θ at run-time to produce a new index environment θ' that is consistent with θ. This relies on sound and complete index substitution

matching (see Section 2.2) to generate θ' such that $\cdot \vdash \theta' : \Delta$ and $\cdot \vdash \Theta[\theta'] = \theta$. We can then evaluate the body t under the new index environment θ' and the same term environment σ to produce a value v.

Notably absent is an evaluation rule for `eq_abort` t. This term is used in a branch of a case split that we know statically to be impossible. Such branches are never reached at run time, so there is no need for an evaluation rule. For example, consider a type-safe "head" function, which receives a nonempty vector as input. As we write each branch of a case split explicitly, the empty list case must use `eq_abort` t, but is never executed. We now state subject reduction for TORES.

▶ **Theorem 7** (Subject Reduction).

1. *If $t[\theta; \sigma] \Downarrow v$ where $\Delta; \cdot; \Gamma \vdash t \Leftarrow T$ or $\Delta; \cdot; \Gamma \vdash t \Rightarrow T$, and $\vdash \theta : \Delta$ and $\sigma : \Gamma[\theta]$, then $v : T[\theta]$.*
2. *If $g[\theta; \sigma] \cdot \vec{N} v \Downarrow w$ where $\Delta; \cdot; \Gamma \vdash g \Leftarrow (\overrightarrow{u{:}U}); S \to T$ and $\vdash \theta : \Delta$ and $\sigma : \Gamma[\theta]$ and $\vdash \vec{N} : (\overrightarrow{u{:}U})[\theta]$ and $v : S[\theta, \overrightarrow{N/u}]$, then $w : T[\theta, \overrightarrow{N/u}]$.*
3. *If $c \cdot_{out_\nu} \Downarrow w$ where $c : (\nu X{:}K. \Lambda \vec{u}. T) \vec{M}$ then $w : T[\overrightarrow{M/u}; (\nu X{:}K. \Lambda \vec{u}. T)/X]$.*

4   Termination Proof

We now describe our main technical result: termination of evaluation. Our proof uses the logical predicate technique of Tait [20] and Girard [10]. We interpret each language construct (index types, kinds, types, etc.) into a semantic model of sets and functions.

4.1   Interpretation of Index Language

We start with the interpretations for index types and spines. In general, our index language may be dependently typed, as it is if we choose Contextual LF. Hence our interpretation for index types U must take into account an environment θ containing instantiations for index variables u. Such an index environment θ is simply a grounding substitution $\vdash \theta : \Delta$.

▶ **Definition 8** (Interpretation of index types $[\![U]\!]$ and index spines $[\![\overrightarrow{u{:}U}]\!]$).

$$
\begin{array}{lll}
[\![U]\!](\theta) & = & \{M \mid \cdot \vdash M : U[\theta]\} \\
[\![(\cdot)]\!](\theta) & = & \{\cdot\} \\
[\![(u_0{:}U_0, \overrightarrow{u{:}U})]\!](\theta) & = & \{M_0, \vec{M} \mid M_0 \in [\![U_0]\!](\theta), \vec{M} \in [\![(\overrightarrow{u{:}U})]\!](\theta, M_0/u_0)\}
\end{array}
$$

The interpretation of an index type U under environment θ is the set of closed terms of type $U[\theta]$. The interpretation lifts to index spines $(\overrightarrow{u{:}U})$. With these definitions, the following lemma follows from the substitution principles of index terms (Req. 3).

▶ **Lemma 9** (Interpretation of index substitution).

9.1. *If $\Delta \vdash M : U$ and $\vdash \theta : \Delta$ then $M[\theta] \in [\![U]\!](\theta)$.*
9.2. *If $\Delta \vdash \vec{M} : (\overrightarrow{u{:}U})$ and $\vdash \theta : \Delta$ then $\overrightarrow{M[\theta]} \in [\![(\overrightarrow{u{:}U})]\!](\theta)$.*

4.2   Lattice Interpretation of Kinds

We now describe the lattice structure that underlies the interpretation of kinds in our language. The idea is that types are interpreted as sets of term-level values and type constructors as functions taking indices to sets of values. We call the set of all term-level values Ω and write its power set as $\mathcal{P}(\Omega)$. The interpretation is defined inductively on the structure of kinds.

▶ **Definition 10** (Interpretation of kinds $[\![K]\!]$).

$$
\begin{array}{rcl}
[\![*]\!](\theta) & = & \mathcal{P}(\Omega) \\
[\![\Pi u{:}U.\,K]\!](\theta) & = & \{\mathcal{C} \mid \forall M \in [\![U]\!](\theta).\, \mathcal{C}(M) \in [\![K]\!](\theta, M/u)\}
\end{array}
$$

A key observation in our metatheory is that each $[\![K]\!](\theta)$ forms a *complete lattice*. In the base case, $[\![*]\!](\theta) = \mathcal{P}(\Omega)$ is a complete lattice under the subset ordering, with meet and join given by intersection and union respectively. For a kind $K = \Pi u{:}U.\,K'$, we induce a lattice structure on $[\![K]\!](\theta)$ by lifting the lattice operations pointwise. Precisely, we define

$$
\mathcal{A} \leq_{[\![K]\!](\theta)} \mathcal{B} \quad \text{iff} \quad \forall M \in [\![U]\!](\theta).\, \mathcal{A}(M) \leq_{[\![K']\!](\theta, M/u)} \mathcal{B}(M).
$$

The meet and join operations can similarly be lifted pointwise.

This structure is important because it allows us to define pre-fixed points for operators on the lattice, which is central to our interpretation of recursive types. Here we rely on the existence of arbitrary meets, as we take the meet over an impredicatively defined subset of $\mathcal{L}$.

▶ **Definition 11** (Mendler-style pre-fixed and post-fixed points). Suppose $\mathcal{L}$ is a complete lattice and $\mathcal{F} : \mathcal{L} \to \mathcal{L}$. Define $\boldsymbol{\mu}_{\mathcal{L}} : (\mathcal{L} \to \mathcal{L}) \to \mathcal{L}$ by

$$
\boldsymbol{\mu}_{\mathcal{L}} \mathcal{F} = \bigwedge \{\mathcal{C} \in \mathcal{L} \mid \forall \mathcal{X} \in \mathcal{L}.\, \mathcal{X} \leq_{\mathcal{L}} \mathcal{C} \implies \mathcal{F}(\mathcal{X}) \leq_{\mathcal{L}} \mathcal{C}\}.
$$

and $\boldsymbol{\nu}_{\mathcal{L}} : (\mathcal{L} \to \mathcal{L}) \to \mathcal{L}$ by

$$
\boldsymbol{\nu}_{\mathcal{L}} \mathcal{F} = \bigvee \{\mathcal{C} \in \mathcal{L} \mid \forall \mathcal{X} \in \mathcal{L}.\, \mathcal{C} \leq_{\mathcal{L}} \mathcal{X} \implies \mathcal{C} \leq_{\mathcal{L}} \mathcal{F}(\mathcal{X})\}.
$$

We will mostly omit the subscript denoting the underlying lattice $\mathcal{L}$ of the order $\leq$ and pre-fixed and post-fixed points, $\boldsymbol{\mu}$ and $\boldsymbol{\nu}$.

Note that a usual treatment of recursive types would define the least pre-fixed point of a monotone operator as $\bigwedge \{\mathcal{C} \in \mathcal{L} \mid \mathcal{F}(\mathcal{C}) \leq \mathcal{C}\}$ and the greatest post-fixed point of a monotone operator as $\bigvee \{\mathcal{C} \in \mathcal{L} \mid \mathcal{C} \leq \mathcal{F}(\mathcal{C})\}$, using the Knaster-Tarski theorem. However, our unconventional definition (following Jacob-Rao et al. [11]) more closely models Mendler-style (co)recursion and does not require $\mathcal{F}$ to be monotone (thereby avoiding a positivity restriction on recursive types).

4.3 Interpretation of Types

In order to interpret the types of our language, it is helpful to define semantic versions of some syntactic constructs. We first define a semantic form of our indexed function type $\overrightarrow{(u{:}U)};\, T_1 \to T_2$, which helps us formulate the interaction of function types with fixed points and recursion.

▶ **Definition 12** (Semantic function space). For a spine interpretation $\vec{\mathcal{U}}$ and functions $\mathcal{A}, \mathcal{B} : \vec{\mathcal{U}} \to \mathcal{P}(\Omega)$, define $\vec{\mathcal{U}}, \mathcal{A} \to \mathcal{B} = \{c \mid \forall \vec{M} \in \vec{\mathcal{U}}.\, \forall v \in \mathcal{A}(\vec{M}).\, c \cdot \vec{M}\, v \Downarrow w \in \mathcal{B}(\vec{M})\}$.

It will also be convenient to lift term-level `in` tags to the level of sets and functions in the lattice $[\![K]\!](\theta)$. We define the lifted tags $\mathtt{in}^* : [\![K]\!](\theta) \to [\![K]\!](\theta)$ inductively on K. If $\mathcal{V} \in [\![*]\!](\theta) = \mathcal{P}(\Omega)$ then $\mathtt{in}^*\,\mathcal{V} = \mathtt{in}\,\mathcal{V} = \{\mathtt{in}\,v \mid v \in \mathcal{V}\}$. If $\mathcal{C} \in [\![\Pi u{:}U.\,K']\!](\theta)$ then $(\mathtt{in}^*\,\mathcal{C})(M) = \mathtt{in}^*\,(\mathcal{C}(M))$ for all $M \in [\![U]\!](\theta)$. Essentially, the $\mathtt{in}^*$ function attaches a tag to every element in the set produced after the index arguments are received.

Dually we define $\mathtt{out}^*_{\nu} : [\![K]\!](\theta) \to [\![K]\!](\theta)$. If $\mathcal{V} \in [\![*]\!](\theta) = \mathcal{P}(\Omega)$ then $\mathtt{out}^*_{\nu}\,\mathcal{V} = \mathtt{out}_{\nu}\,\mathcal{V} = \{c \mid c \cdot_{\mathtt{out}_{\nu}} \Downarrow w \in \mathcal{V}\}$. If $\mathcal{C} \in [\![\Pi u{:}U.\,K']\!](\theta)$ then $(\mathtt{out}^*_{\nu}\,\mathcal{C})(M) = \mathtt{out}^*_{\nu}\,(\mathcal{C}(M))$ for all $M \in [\![U]\!](\theta)$.

Finally, we define the interpretation of type variable contexts Ξ. These describe semantic environments η mapping each type variable to an object in its respective kind interpretation. Such environments are necessary to interpret type expressions with free type variables.

▶ **Definition 13** (Interpretation of type variable contexts $[\![\Xi]\!]$).

$$
\begin{aligned}
[\![\cdot]\!](\theta) &= \{\cdot\} \\
[\![\Xi, X{:}K]\!](\theta) &= \{\eta, \mathcal{X}/X \mid \eta \in [\![\Xi]\!](\theta), \mathcal{X} \in [\![K]\!](\theta)\}
\end{aligned}
$$

We are now able to define the interpretation of types T under environments θ and η. This is done inductively on the structure of T.

▶ **Definition 14** (Interpretation of types and constructors).

$$
\begin{aligned}
[\![1]\!](\theta;\eta) &= \{\langle\rangle\} \\
[\![T_1 \times T_2]\!](\theta;\eta) &= \{\langle v_1, v_2\rangle \mid v_1 \in [\![T_1]\!](\theta;\eta), v_2 \in [\![T_2]\!](\theta;\eta)\} \\
[\![T_1 + T_2]\!](\theta;\eta) &= \mathtt{in}_1\,[\![T_1]\!](\theta;\eta) \bigcup \mathtt{in}_2\,[\![T_2]\!](\theta;\eta) \\
[\![\overrightarrow{(u{:}U)}; T_1 \to T_2]\!](\theta;\eta) &= [\![\overrightarrow{(u{:}U)}]\!](\theta), \mathcal{T}_1 \to \mathcal{T}_2 \\
&\quad \text{where } \mathcal{T}_i(\vec{M}) = [\![T_i]\!](\theta, \overrightarrow{M/u};\eta) \text{ for } i \in \{1,2\} \\
[\![\Sigma u{:}U.\,T]\!](\theta;\eta) &= \{\mathtt{pack}\,(M,v) \mid M \in [\![U]\!](\theta), v \in [\![T]\!](\theta, M/u;\eta)\} \\
[\![T\,M]\!](\theta;\eta) &= [\![T]\!](\theta;\eta)(M[\theta]) \\
[\![M = N]\!](\theta;\eta) &= \{\mathtt{refl} \mid \;\vdash M[\theta] = N[\theta]\} \\
[\![X]\!](\theta;\eta) &= \eta(X) \\
[\![\Lambda\,u.\,T]\!](\theta;\eta) &= (M \mapsto [\![T]\!](\theta, M/u;\eta)) \\
[\![\mu X{:}K.\,T]\!](\theta;\eta) &= \boldsymbol{\mu}_{[\![K]\!](\theta)}(\mathcal{X} \mapsto \mathtt{in}_\mu^*\,([\![T]\!](\theta;\eta, \mathcal{X}/X))) \\
[\![\nu X{:}K.\,T]\!](\theta;\eta) &= \boldsymbol{\nu}_{[\![K]\!](\theta)}(\mathcal{X} \mapsto \mathtt{out}_\nu^*\,([\![T]\!](\theta;\eta, \mathcal{X}/X))) \\
[\![\mathtt{Rec}_K\,(0 \mapsto T_z \mid \mathtt{suc}\,u,\,X \mapsto T_s)]\!](\theta;\eta) &= \mathbf{Rec}_{[\![K]\!](\theta)}\,(\mathtt{in}_0^*\,[\![T_z]\!](\theta;\eta)) \\
&\qquad (N \mapsto \mathcal{X} \mapsto \mathtt{in}_{\mathsf{suc}}^*\,[\![T_s]\!](\theta, N/u;\eta, \mathcal{X}/X))
\end{aligned}
$$

where

$$
\begin{aligned}
\mathbf{Rec}_\mathcal{L} : \quad &\mathcal{L} \to \quad (\mathbb{N} \to \mathcal{L} \to \mathcal{L}) \to \quad \mathbb{N} \qquad \to \quad \mathcal{L} \\
\mathbf{Rec}_\mathcal{L} \quad &\mathcal{C} \qquad \mathcal{F} \qquad\qquad\qquad\qquad 0 \qquad\quad = \quad \mathcal{C} \\
\mathbf{Rec}_\mathcal{L} \quad &\mathcal{C} \qquad \mathcal{F} \qquad\qquad\qquad (\mathtt{suc}\,N) \;\; = \quad \mathcal{F}\,N\,(\mathbf{Rec}_\mathcal{L}\,\mathcal{C}\,\mathcal{F}\,N)
\end{aligned}
$$

The interpretation of the indexed function type $[\![\overrightarrow{(u{:}U)}; T_1 \to T_2]\!](\theta;\eta)$ contains closures which, when applied to values in the appropriate input sets, evaluate to values in the appropriate output set. The interpretation of the equality type $[\![M = N]\!](\theta;\eta)$ is the set $\{\mathtt{refl}\}$ if $\vdash M[\theta] = N[\theta]$ and the empty set otherwise. The interpretation of a recursive type is the pre-fixed point of the function obtained from the underlying type expression. Finally, interpretation of a stratified type built from $\mathtt{Rec}$ relies on an analogous semantic operator $\mathbf{Rec}$. It is defined by primitive recursion on the index argument, returning the first argument in the base case and calling itself recursively in the step case. Note that the definition of $\mathbf{Rec}$ is specific to the index type it recurses over. We only use the index language of natural numbers here, so the appropriate set of index values is $[\![\mathsf{nat}]\!] = \mathbb{N}$.

Last, we give the interpretation for typing contexts Γ, describing well-formed term-level environments σ.

▶ **Definition 15** (Interpretation of typing contexts).

$$
\begin{aligned}
[\![\cdot]\!](\theta;\eta) &= \{\cdot\} \\
[\![\Gamma, x{:}T]\!](\theta;\eta) &= \{\sigma, v/x \mid \sigma \in [\![\Gamma]\!](\theta;\eta), v \in [\![T]\!](\theta;\eta)\}
\end{aligned}
$$

4.4 Proof

We now sketch our proof using some key lemmas. The following two lemmas concern the fixed point operators $\boldsymbol{\mu}$ and $\boldsymbol{\nu}$, and are key for reasoning about (co)recursive types and Mendler-style (co)recursion. These lemmas generalize those of Jacob-Rao et al. [11] from the simply typed setting.

▶ **Lemma 16** (Soundness of pre-fixed point). *Suppose $\mathcal{L}$ is a complete lattice, $\mathcal{F} : \mathcal{L} \to \mathcal{L}$ and $\boldsymbol{\mu}$ is as in Def. 11. Then $\mathcal{F}(\boldsymbol{\mu}\mathcal{F}) \leq \boldsymbol{\mu}\mathcal{F}$.*

▶ **Lemma 17** (Function space from pre-fixed and post-fixed points). *Let $\mathcal{L} = \vec{\mathcal{U}} \to \mathcal{P}(\Omega)$ and $\mathcal{B} \in \mathcal{L}$ and $\mathcal{F} : \mathcal{L} \to \mathcal{L}$.*

1. *If $\forall \mathcal{X} \in \mathcal{L}. \ c \in \vec{\mathcal{U}}, \mathcal{X} \to \mathcal{B} \implies c \in \vec{\mathcal{U}}, \mathcal{F}\mathcal{X} \to \mathcal{B}$, then $c \in \vec{\mathcal{U}}, \boldsymbol{\mu}\mathcal{F} \to \mathcal{B}$.*
2. *If $\forall \mathcal{X} \in \mathcal{L}. \ c \in \vec{\mathcal{U}}, \mathcal{B} \to \mathcal{X} \implies c \in \vec{\mathcal{U}}, \mathcal{B} \to \mathcal{F}\mathcal{X}$, then $c \in \vec{\mathcal{U}}, \mathcal{B} \to \boldsymbol{\nu}\mathcal{F}$.*

Another key result we rely on is that type-level substitutions associate with our semantic interpretations. Note that single index (and spine) substitutions on types are handled as special cases of the result for simultaneous index substitutions. We omit the definitions of type substitutions for brevity.

▶ **Lemma 18** (Type-level substitution associates with interpretation).
Suppose $\Delta; \Xi \vdash T \Leftarrow K$ or $\Delta; \Xi \vdash T \Rightarrow K$, and $\vdash \theta : \Delta'$ and $\eta \in [\![\Xi']\!](\theta)$.

1. *If $\Delta' \vdash \Theta : \Delta$ and $\Xi' = \Xi[\Theta]$ then $[\![\Xi']\!](\theta) = [\![\Xi]\!](\Theta[\theta])$ and $[\![T[\Theta]]\!](\theta; \eta) = [\![T]\!](\Theta[\theta]; \eta)$.*
2. *If $\Delta = \Delta'$ and $\Xi = \Xi', X{:}K$ and $\Delta'; \Xi' \vdash S \Leftarrow K$ or $\Delta'; \Xi' \vdash S \Rightarrow K$, then $[\![T[S/X]]\!](\theta; \eta) = [\![T]\!](\theta; \eta, [\![S]\!](\theta; \eta)/X)$.*

Proof. By induction on the structure of T.   ◀

The next two lemmas concern recursive types and terms respectively.

▶ **Lemma 19** (Recursive type contains unfolding).
Let $R = \mu X{:}K. \Lambda \vec{u}. S$ where $K = \Pi\overrightarrow{u{:}U}. *$ and $\Delta; \Xi \vdash R \Rightarrow K$, and $\Delta \vdash \vec{M} : (\overrightarrow{u{:}U})$ and $\vdash \theta : \Delta$ and $\eta \in [\![\Xi]\!](\theta)$. Then $\boldsymbol{in}_\mu [\![S[\overrightarrow{M/u}; R/X]]\!](\theta; \eta) \subseteq [\![R \, \vec{M}]\!](\theta; \eta)$.

▶ **Lemma 20** (Backward closure).
Let t be a term, θ and σ environments, and $\mathcal{A}, \mathcal{B} \in \vec{\mathcal{U}} \to \mathcal{P}(\Omega)$.

1. *If $t[\theta; \sigma, (\mathtt{rec} \, f. \, t)[\theta; \sigma]/f] \Downarrow c' \in \vec{\mathcal{U}}, \mathcal{A} \to \mathcal{B}$, then $(\mathtt{rec} \, f. \, t)[\theta; \sigma] \in \vec{\mathcal{U}}, \boldsymbol{in}_\mu^* \mathcal{A} \to \mathcal{B}$.*
2. *If $t[\theta; \sigma, (\mathtt{corec} \, f. \, t)[\theta; \sigma]/f] \Downarrow c' \in \vec{\mathcal{U}}, \mathcal{A} \to \mathcal{B}$, then $(\mathtt{corec} \, f. \, t)[\theta; \sigma] \in \vec{\mathcal{U}}, \mathcal{A} \to \boldsymbol{out}_\nu^* \mathcal{B}$.*

Our final lemma concerns the semantic equivalence of an applied stratified type with its unfolding. Note that here we only state and prove the lemma for an index language of natural numbers. For a different index language, one would need to reverify this lemma for the corresponding stratified type. This should be straight-forward once the semantic **Rec** operator is chosen to reflect the inductive structure of the index language.

▶ **Lemma 21** (Stratified types equivalent to unfolding).
Let $T_{\mathtt{Rec}} \equiv \mathtt{Rec}_K (0 \mapsto T_z \mid \mathtt{suc} \, n, X \mapsto T_s)$ where $K = \Pi n{:}\mathsf{nat} . \Pi \overrightarrow{u{:}U}. *$ and $\Delta; \Xi \vdash T_{\mathtt{Rec}} \Rightarrow K$, and $\Delta \vdash \vec{M} : (\overrightarrow{u{:}U})$ and $\Delta \vdash N : \mathsf{nat}$ and $\vdash \theta : \Delta$ and $\eta \in [\![\Xi]\!](\theta)$. Then

1. $[\![T_{\mathtt{Rec}} \, 0 \, \vec{M}]\!](\theta; \eta) = \boldsymbol{in}_0 ([\![T_z \, \vec{M}]\!](\theta; \eta))$ *and*
2. $[\![T_{\mathtt{Rec}} (\mathtt{suc} \, N) \, \vec{M}]\!](\theta; \eta) = \boldsymbol{in}_{\mathtt{suc}} ([\![T_s[N/n; (T_{\mathtt{Rec}} \, N)/X] \, \vec{M}]\!](\theta; \eta))$.

Finally we state the main termination theorem.

▶ **Theorem 22** (Termination of evaluation). *If $\Delta; \Xi; \Gamma \vdash t \Leftarrow T$ or $\Delta; \Xi; \Gamma \vdash t \Rightarrow T$, and $\vdash \theta : \Delta$ and $\eta \in [\![\Xi]\!](\theta)$ and $\sigma \in [\![\Gamma]\!](\theta; \eta)$, then $t[\theta; \sigma] \Downarrow v$ for some $v \in [\![T]\!](\theta; \eta)$.*

5 Related Work

Our work draws inspiration from two different areas: dependent type theory and proof theory.

Dependent type theories often support large eliminations that are definitions of dependent types by primitive recursion. They reduce the same way as term-level recursion. In general, our work shows how to gain the power of large eliminations in an indexed type system by simulating reduction on the level of types by unfolding stratified types in their typing rules.

In the world of proof theory, our core language corresponds to a first-order logic with equality, inductive and stratified (recursive) definitions. Momigliano and Tiu [14, 23] give comprehensive treatments of logics with induction and co-induction as well as first-class equality. They present their logics in a sequent calculus style and prove cut elimination which implies consistency of the logics. Their cut elimination proof extends Girard's proof technique of reducibility candidates, similar to ours. Note that they require strict positivity of inductive definitions, i.e. the head of a definition (analogous to the recursive type variable) is not allowed to occur to the left of an implication.

Tiu [22] also develops a first-order logic with stratified definitions similar to our stratified types. His notion of stratification comes from defining the "level" of a formula, which measures its size. A recursive definition is then called stratified if the level does not increase from the head of the definition to the body. This is a more general formulation than our notion of stratification for types: we require the type to be stratified exactly according to the structure of an index term, instead of a more general decreasing measure. However, we could potentially replicate such a measure by suitably extending our index language.

Another approach to supporting recursive definitions in proof theory is via a rewriting relation, as in the Deduction Modulo system [8]. The idea of this system is to generalize a given first-order logic to account for a congruence relation defined by a set of rewrite rules. This rewriting could include recursive definitions in the same sense as Tiu. Dowek and Werner [9] show that such logics under congruences can be proven normalizing given general conditions on the congruence. Baelde and Nadathur [3] extend this work in the following way. First, they present a first-order logic with inductive and co-inductive definitions, together with a general form of equality. They show strong normalization for this logic using a reducibility candidate argument. Crucially, their proof is in terms of a pre-model which anticipates the addition of recursive definitions via a rewrite relation. Then they give conditions on the rewrite rules, essentially requiring that each definition follows a well-founded order on its arguments. Under these conditions, they are able to construct a pre-model for the relation, proving normalization as a result. Although their notion of stratification of recursive definitions is slightly more general than ours, our treatment is perhaps more direct as the rewriting of types takes place within our typing rules, and our semantic model accounts for stratified types directly.

From a programmer's view, the proof theoretic foundations give rise to programs written using iterators; our use of Mendler-style (co)recursion is arguably closer to standard programming practice. Mendler-style recursion schemes for term-indexed languages have been investigated by Ahn [2]. Ahn describes an extension of System F_ω with erasable term indices, called F_i. He combines this with fixed points of type operators, as in the Fix_ω language by Abel and Matthes [1], to produce the core language Fix_i. In Fix_i, one can embed Mendler-style recursion over term-indexed data types by Church-style encodings.

Fundamentally, our use of indices is more liberal than in Ahn's core languages. In F_i, term indices are drawn from the same term language as programs. They are treated polymorphically, in analogue with polymorphic type indices, i.e. they must have closed types and cannot be analyzed at runtime. Our approach is to separate the language of index terms

from the language of programs. In Tores, the indices that appear in types can be handled and analyzed at runtime, may be dependently typed and have types with free variables. This flexibility is crucial for writing inductive proofs over LF specifications as we do in Beluga.

6 Conclusion and Future Work

We presented a core language Tores extending an indexed type system with (co)recursive types and stratified types. We argued that Tores provides a sound and powerful foundation for programming (co)inductive proofs, in particular those involving logical relations. This power comes from the (co)induction principles on recursive types given by Mendler-style (co)recursion as well as the flexibility of recursive definitions given by stratified types. Type checking in Tores is decidable and types are preserved during evaluation. The soundness of our language is guaranteed by our logical predicate semantics and termination proof. We believe that Tores balances well the proof-theoretic power with a simple metatheory (especially when compared with full dependent types).

An important question to investigate in the future is how to compile a practical language that supports (co)pattern matching into the core language we propose in Tores. Similarly it would also be interesting to explore how our treatment of indexed recursive and stratified types could help (or hinder) proof search. Such issues are important to solve in order to create a productive user experience for dependently typed programming and proving.

References

1 Andreas Abel and Ralph Matthes. Fixed points of type constructors and primitive recursion. In *18th Int'l Workshop on Computer Science Logic (CSL'04)*, Lecture Notes in Computer Science (LNCS 3210), pages 190–204. Springer, 2004.

2 Ki Yung Ahn. *The Nax Language: Unifying Functional Programming and Logical Reasoning in a Language based on Mendler-style Recursion Schemes and Term-indexed Types*. PhD thesis, Portland State University, 2014.

3 David Baelde and Gopalan Nadathur. Combining deduction modulo and logics of fixed-point definitions. In *27th Annual IEEE Symp. on Logic in Computer Science (LICS'12)*, pages 105–114. IEEE, 2012.

4 Andrew Cave and Brigitte Pientka. Programming with binders and indexed data-types. In *39th ACM Symp. on Principles of Programming Languages (POPL'12)*, pages 413–424. ACM, 2012.

5 Andrew Cave and Brigitte Pientka. First-class substitutions in contextual type theory. In *8th ACM Int'l Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'13)*, pages 15–24. ACM, 2013.

6 Andrew Cave and Brigitte Pientka. A case study on logical relations using contextual types. In *10th Int'l Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'15)*, pages 18–33. Electronic Proceedings in Theoretical Computer Science (EPTCS), 2015.

7 James Cheney and Ralf Hinze. First-class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, 2003.

8 Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31(1):33–72, 2003.

9 Gilles Dowek and Benjamin Werner. Proof normalization modulo. *J. Symb. Log.*, 68(4):1289–1316, 2003.

10 J. Y Girard. *Interprétation fonctionnelle et elimination des coupures de l'arithmétique d'ordre supérieur*. These d'état, Université de Paris 7, 1972.

11   Rohan Jacob-Rao, Andrew Cave, and Brigitte Pientka. Mechanizing proofs about Mendler-style recursion. In *11th Int'l Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'16)*, pages 1–9. ACM, 2016.

12   Raymond C. McDowell and Dale A. Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Transactions on Computational Logic*, 3(1):80–136, 2002.

13   Nax Paul Francis Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, Ithaca, NY, USA, 1988. AAI8804634.

14   Alberto Momigliano and Alwen Fernanto Tiu. Induction and co-induction in sequent calculus. In *Types for Proofs and Programs (TYPES'03)*, Lecture Notes in Computer Science (LNCS 3085), pages 293–308. Springer, 2004.

15   Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3):1–49, 2008.

16   Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *35th ACM Symp. on Principles of Programming Languages (POPL'08)*, pages 371–382. ACM, 2008.

17   Brigitte Pientka and Andreas Abel. Structural recursion over contextual objects. In *13th Int'l Conf. on Typed Lambda Calculi and Applications (TLCA'15)*, pages 273–287. Leibniz Int'l Proceedings in Informatics (LIPIcs) of Schloss Dagstuhl, 2015.

18   Brigitte Pientka and Joshua Dunfield. Programming with proofs and explicit contexts. In *35th ACM Symp. on Principles and Practice of Declarative Programming (PPDP'08)*, pages 163–173. ACM, 2008.

19   Peter Schroeder-Heister. Rules of definitional reflection. In *8th Annual Symp. on Logic in Computer Science (LICS '93)*, pages 222–232. IEEE Computer Society, 1993.

20   William Tait. Intensional Interpretations of Functionals of Finite Type I. *J. Symb. Log.*, 32(2):198–212, 1967.

21   David Thibodeau, Andrew Cave, and Brigitte Pientka. Indexed codata. In *21st ACM Int'l Conf. on Functional Programming (ICFP'16)*, pages 351–363. ACM, 2016.

22   Alwen Tiu. Stratification in logics of definitions. In *6th Int'l Joint Conf. on Automated Reasoning (IJCAR'12)*, Lecture Notes in Computer Science (LNCS 7364), pages 544–558. Springer, 2012.

23   Alwen Tiu and Alberto Momigliano. Cut elimination for a logic with induction and co-induction. *J. Applied Logic*, 10(4):330–367, 2012.

24   Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *30th ACM Symp. on Principles of Programming Languages (POPL'03)*, pages 224–235. ACM, 2003.

A   Appendix

A.1   Kinding

We employ a bidirectional kinding system to show when kinds can be inferred and when kinding annotations are necessary. The judgments use two contexts: an index context Δ and a type variable context Ξ. Note that in general the kinds assigned to type variables in Ξ may depend on index variables in Δ.

A.2   Value Typing

Values are the results of evaluation. Note that values are closed, and hence their typing judgment does not require a context. However, closures do contain terms (typed with the main typing judgment) and environments (typed against the contexts Δ and Γ).

$$\boxed{\Delta; \Xi \vdash T \Leftarrow K} \qquad \text{Check type } T \text{ against kind } K$$

$$\frac{}{\Delta; \Xi \vdash 1 \Leftarrow *} \qquad \frac{\Delta; \Xi \vdash T_1 \Leftarrow * \quad \Delta; \Xi \vdash T_2 \Leftarrow *}{\Delta; \Xi \vdash T_1 \times T_2 \Leftarrow *} \qquad \frac{\Delta; \Xi \vdash T_1 \Leftarrow * \quad \Delta; \Xi \vdash T_2 \Leftarrow *}{\Delta; \Xi \vdash T_1 + T_2 \Leftarrow *}$$

$$\frac{\Delta \vdash (\overrightarrow{u{:}U}) \text{ itype} \quad \Delta, \overrightarrow{u{:}U}; \Xi \vdash S \Leftarrow * \quad \Delta, \overrightarrow{u{:}U}; \Xi \vdash T \Leftarrow *}{\Delta; \Xi \vdash (\overrightarrow{u{:}U}); S \to T \Leftarrow *} \qquad \frac{\Delta \vdash U \text{ itype} \quad \Delta, u{:}U; \Xi \vdash T \Leftarrow *}{\Delta; \Xi \vdash \Sigma u{:}U. T \Leftarrow *}$$

$$\frac{\Delta \vdash M : U \quad \Delta \vdash N : U}{\Delta; \Xi \vdash M = N \Leftarrow *} \qquad \frac{\Delta, u{:}U; \Xi \vdash T \Leftarrow K}{\Delta; \Xi \vdash \Lambda\, u. T \Leftarrow \Pi u{:}U. K} \qquad \frac{\Delta; \Xi \vdash T \Rightarrow *}{\Delta; \Xi \vdash T \Leftarrow *}$$

$$\boxed{\Delta; \Xi \vdash T \Rightarrow K} \qquad \text{Infer a kind } K \text{ for type } T$$

$$\frac{X{:}K \in \Xi}{\Delta; \Xi \vdash X \Rightarrow K} \qquad \frac{\Delta; \Xi \vdash T \Rightarrow \Pi u{:}U. K \quad \Delta \vdash M : U}{\Delta; \Xi \vdash T\, M \Rightarrow K[M/u]}$$

$$\frac{\Delta; \Xi, X{:}K \vdash T \Leftarrow K}{\Delta; \Xi \vdash \mu X{:}K. T \Rightarrow K} \qquad \frac{\Delta; \Xi, X{:}K \vdash T \Leftarrow K}{\Delta; \Xi \vdash \nu X{:}K. T \Rightarrow K}$$

$$\frac{K = \Pi u{:}\, \mathsf{nat}\, . \, K' \quad \Delta; \Xi \vdash T_0 \Leftarrow K'[0/u] \quad \Delta, u{:}\mathsf{nat}; \Xi, X{:}K' \vdash T_s \Leftarrow K'[\mathsf{suc}\, u/u]}{\Delta; \Xi \vdash \mathtt{Rec}_K(0 \mapsto T_0 \mid \mathsf{suc}\, u,\, X \mapsto T_s) \Rightarrow K}$$

Figure 4 Kinding rules for TORES.

$$\boxed{v : T} \qquad \text{Value } v \text{ has type } T$$

$$\frac{\cdot \vdash \theta : \Delta \quad \sigma : \Gamma[\theta] \quad \Delta; \cdot; \Gamma \vdash g \Leftarrow T}{(g)[\theta; \sigma] : T[\theta]} \qquad \frac{}{\langle\rangle : 1} \qquad \frac{\cdot \vdash M = N}{\mathtt{refl} : M = N} \qquad \frac{v_1 : T_1 \quad v_2 : T_2}{\langle v_1, v_2 \rangle : T_1 \times T_2}$$

$$\frac{v : T_i}{\mathtt{in}_i\, v : T_1 + T_2} \qquad \frac{\cdot \vdash M : U \quad v : T[M/u]}{\mathtt{pack}\,(M, v) : \Sigma u{:}U. T} \qquad \frac{v : T[\overrightarrow{M/u}; \mu X{:}K.\, \Lambda\, \vec{u}.\, T/X]}{\mathtt{in}_\mu\, v : (\mu X{:}K.\, \Lambda\, \vec{u}.\, T)\, \vec{M}}$$

$$\frac{v : T_0\, \vec{M}}{\mathtt{in}_0\, v : T_{\mathtt{Rec}}\, 0\, \vec{M}} \qquad \frac{v : T_s[N/u; T_{\mathtt{Rec}}\, N/X]\, \vec{M}}{\mathtt{in}_{\mathsf{suc}}\, v : T_{\mathtt{Rec}}\, (\mathsf{suc}\, N) \vec{M}}$$

$$\frac{\cdot \vdash \theta : \Delta \quad \sigma : \Gamma[\theta] \quad \cdot \vdash \vec{N} : \vec{U} \quad v : S[\theta, \overrightarrow{N/u}]}{\Delta; \cdot, X{:}K; \Gamma, f{:}((\overrightarrow{u{:}U}); S \to X\vec{u}) \vdash t \Leftarrow (\overrightarrow{u{:}U}); S \to T[\overrightarrow{u/u'}]}{(\mathtt{corec}\, f.\, t)[\theta; \sigma] \cdot \vec{N}\; v : (\nu X{:}K.\, \Lambda\, \overrightarrow{u'}.\, T)[\theta]\; \vec{N}}$$

$$\boxed{\sigma : \Gamma} \qquad \text{Environment } \sigma \text{ has domain } \Gamma$$

$$\frac{}{\cdot : \cdot} \qquad \frac{\sigma : \Gamma \quad v : T}{(\sigma, v/x) : \Gamma, x{:}T}$$

Figure 5 Value and environment typing.

A Syntax for Higher Inductive-Inductive Types

Ambrus Kaposi

Faculty of Informatics, Eötvös Loránd University, Pázmány Péter sétány 1/C, 1117 Budapest, Hungary
akaposi@inf.elte.hu
🆔 https://orcid.org/0000-0001-9897-8936

András Kovács

Faculty of Informatics, Eötvös Loránd University, Pázmány Péter sétány 1/C, 1117 Budapest, Hungary
kovacsandras@inf.elte.hu
🆔 https://orcid.org/0000-0002-6375-9781

──── **Abstract** ────

Higher inductive-inductive types (HIITs) generalise inductive types of dependent type theories in two directions. On the one hand they allow the simultaneous definition of multiple sorts that can be indexed over each other. On the other hand they support equality constructors, thus generalising higher inductive types of homotopy type theory. Examples that make use of both features are the Cauchy reals and the well-typed syntax of type theory where conversion rules are given as equality constructors. In this paper we propose a general definition of HIITs using a domain-specific type theory. A context in this small type theory encodes a HIIT by listing the type formation rules and constructors. The type of the elimination principle and its β-rules are computed from the context using a variant of the syntactic logical relation translation. We show that for indexed W-types and various examples of HIITs the computed elimination principles are the expected ones. Showing that the thus specified HIITs exist is left as future work. The type theory specifying HIITs was formalised in Agda together with the syntactic translations. A Haskell implementation converts the types of sorts and constructors into valid Agda code which postulates the elimination principles and computation rules.

1   Introduction

Many dependent type theories support some form of inductive types. An inductive type is given by its constructors, along with an elimination principle which expresses that all inhabitants are constructed using finitely many applications of the constructors.

For example, the inductive type of natural numbers Nat is given by the constructors zero : Nat and suc : Nat $\to$ Nat. The eliminator corresponds to the usual notion of mathematical induction:

$$\mathsf{ElimNat} : (P : \mathsf{Nat} \to \mathsf{Type})(pz : P\,\mathsf{zero})\big(ps : (n : \mathsf{Nat}) \to P\,n \to P\,(\mathsf{suc}\,n)\big)(n : \mathsf{Nat}) \to P\,n$$

P is a family of types over natural numbers, which is called the *motive* of the eliminator. It can be viewed as a proof-relevant predicate on Nat. The arguments pz and ps are called the *methods* of the eliminator. The *target* of the eliminator is n and given methods for each constructor, the eliminator provides a witness of $P\,n$. Thus, if P holds for zero and suc preserves P, then P holds for all natural numbers. The behaviour of the eliminator is described by a *computation-rule* (β-rule) for each constructor:

$$\mathsf{ElimNat}\,P\,pz\,ps\,\mathsf{zero} \quad \equiv pz$$
$$\mathsf{ElimNat}\,P\,pz\,ps\,(\mathsf{suc}\,n) \equiv ps\,n\,(\mathsf{ElimNat}\,P\,pz\,ps\,n)$$

These express that the eliminator applied to a constructor expression reduces to an application of the corresponding induction method. From an operational point of view, ElimNat replaces all the zero and suc constructors with the given induction methods.

Dependent families of types can be defined in a similar way, for example vectors of A-elements $\mathsf{Vec}_A : \mathsf{Nat} \to \mathsf{Type}$ which are indexed by their length. Another generalisation of inductive types are mutual inductive types. However, these can be reduced to indexed families where indices classify constructors for each mutual type. Inductive-inductive types [23] are mutual definitions where this reduction does not work: here a type is defined together with a family indexed over it. An example is the following fragment of the well-typed syntax of type theory where the second sort Ty is indexed over the first sort Con, but constructors of Con also refer to Ty:

Con	: Type	sort of contexts
Ty	: Con $\to$ Type	sort of types given a context
$\bullet$	: Con	constructor for the empty context
$-\rhd-$	: $(\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Con}$	constructor for context extension
U	: $(\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma$	constructor for a base type
Π	: $(\Gamma : \mathsf{Con})(A : \mathsf{Ty}\,\Gamma) \to \mathsf{Ty}\,(\Gamma \rhd A) \to \mathsf{Ty}\,\Gamma$	constructor for dependent functions

There are two eliminators for this type: one for Con and one for Ty. Both take the same arguments: two motives ($P : \mathsf{Con} \to \mathsf{Type}$ and $Q : (\Gamma : \mathsf{Con}) \to P\,\Gamma \to \mathsf{Ty}\,\Gamma \to \mathsf{Type}$) and four methods (one for each constructor, we don't list these).

$$\mathsf{ElimCon} : (P : \dots)(Q : \dots) \to \dots \to (\Gamma : \mathsf{Con})\ \to P\,\Gamma$$
$$\mathsf{ElimTy}\ : (P : \dots)(Q : \dots) \to \dots \to (A : \mathsf{Ty}\,\Gamma) \to Q\,\Gamma\,(\mathsf{ElimCon}\,\Gamma)\,A$$

Note that the type of ElimTy refers to ElimCon, which is why this elimination principle is called recursive-recursive (analogously to the phrase "inductive-inductive").

Higher inductive types (HITs, [25, Chapter 6]) generalise inductive types in a different way: they allow constructors expressing equalities of elements of the type being defined. This enables, among others, the definition of types quotiented by a relation. For example, the type of integers Int can be given by a constructor pair : Nat $\to$ Nat $\to$ Int and an equality constructor eq : $(a\,b\,c\,d : \mathsf{Nat}) \to a + d =_{\mathsf{Nat}} b + c \to \mathsf{pair}\,a\,b =_{\mathsf{Int}} \mathsf{pair}\,c\,d$ targetting an

equality of Int. The eliminator for Int expects a motive $P : \text{Int} \to \text{Type}$, a method for the pair constructor $p : (a\, b : \text{Nat}) \to P\,(\text{pair}\, a\, b)$ and a method for the equality constructor path. This method is a proof that given $e : a + d =_{\text{Nat}} b + c$, $p\, a\, b$ is equal to $p\, c\, d$ (the types of which are equal by e). Thus the method for the equality constructor ensures that all functions defined from the quotiented type respect the relation. Since the integers are supposed to be a set (which means that any two equalities between the same two integers are equal), we would need an additional higher equality constructor $\text{trunc} : (x\, y : \text{Int}) \to (p\, q : x =_{\text{Int}} y) \to p =_{x=_{\text{Int}}y} q$. HITs allow equality constructors at any level. With the view of types as spaces in mind, point constructors add points to the space, equality constructors add paths and higher constructors add homotopies between paths.

Not all constructor expressions make sense. For example [25, Example 6.13.1], given an $f : (X : \text{Type}) \to X \to X$, suppose that an inductive type Ival is generated by the point constructors $\text{a} : \text{Ival}$, $\text{b} : \text{Ival}$ and a path constructor $\sigma : f\, \text{Ival}\, \text{a} =_{\text{Ival}} f\, \text{Ival}\, \text{b}$. The eliminator for this type should take a motive $P : \text{Ival} \to \text{Type}$, two methods $p_a : P\, \text{a}$ and $p_b : P\, \text{b}$, and a path connecting elements of $P\,(f\, \text{Ival}\, \text{a})$ and $P\,(f\, \text{Ival}\, \text{b})$. However it is not clear what these elements should be: we only have elements $p_a : P\, \text{a}$ and $p_b : P\, \text{b}$, and there is no way in general to transform these to have types $P\,(f\, \text{Ival}\, \text{a})$ and $P\,(f\, \text{Ival}\, \text{b})$.

Another invalid example is an inductive type Neg with a constructor $\text{con} : (\text{Neg} \to \bot) \to \text{Neg}$ where $\bot$ is the empty type. An eliminator for this type should (at least) yield a projection function $\text{proj} : \text{Neg} \to (\text{Neg} \to \bot)$. Given this, we can define $u :\equiv \text{con}\,(\lambda x.\text{proj}\, x\, x) : \text{Neg}$ and then derive $\bot$ by $\text{proj}\, u\, u$. The existence of Neg would make the type theory inconsistent. A common restriction to avoid such situations is *strict positivity*. It means that the type being defined cannot occur on the left hand side of a function arrow in a parameter of a constructor. This excludes the above constructor con.

In this paper we propose a general syntax for higher inductive-inductive types (HIITs) which includes the above positive examples and excludes the negative ones. Our syntax for HIITs allows any number of inductive-inductive sorts, possibly infinitary higher constructors of any dimension and restricts constructors to strictly positive ones. It also allows free usage of J and refl in HIIT specifications. We also show how to derive the types of the eliminators and computation rules from the type formation rules and constructors.

The core idea is to represent HIIT specifications as contexts in a domain-specific type theory which we call the *theory of codes*. A context in this theory can be seen as a *code* for a HIIT, similarly to how a container [1] can be seen as a code for a simple inductive type. Type formers in the theory of codes are restricted in order to enforce strict positivity. For example, natural numbers are defined as the three-element context

$$Nat : \mathsf{U}, \ \ zero : \underline{Nat}, \ \ suc : Nat \to \underline{Nat}$$

where Nat, $zero$ and suc are simply variable names, and underlining denotes El (decoding) for the Tarski-style universe U.

We use a variant of Bernardy et al.'s logical predicate translation [8] to derive the types of motives and methods, and a logical relation translation to derive the types of the eliminators and computation rules. The target of these translations is a type theory with a predicative hierarchy of Russell-style universes closed under Π, Σ, the equality (identity) type $- = -$ and the unit type $\top$. The source type theory is the target type theory extended with rules for the theory of codes.

To our knowledge, this is the first proposal for a definition of HIITs. Proving the existence of the HIITs thus specified is left as future work.

1.1   Overview of the paper

We start by describing the target type theory in Section 2. In Section 3, we define the source type theory. The source type theory is the target theory extended with the theory of codes, i.e. the rules to describe HIITs. We also provide several examples of HIIT definitions. In Section 4 we define three syntactic translations from the source to the target theory, each depending on the previous one: first, we compute the types of type formation rules and constructors (Section 4.1); then, assuming the constructors exist, we compute the types of motives and methods (Section 4.2); finally we compute the types of the eliminators together with their computation rules (Section 4.3). To illustrate these operations, we show how they compute on a few example codes: natural numbers, the circle, indexed W-types and the two-dimensional sphere (Appendix A). In Section 5 we add the pieces together by specifying what it means for the target type theory to support HIITs. Section 6 describes the formalisation and a Haskell implementation. We conclude in Section 7.

1.2   Related work

Inductive types can be specified using external syntactic schemes or internal codes. In the former case the type theory is extended with derivation rules specifying inductive types. In the latter case there is an internal type of codes such that each code represents a valid inductive type, and actual types are produced from codes by decoding functions. Our development uses the former approach.

External schemes for inductive families are given in [13, 24], for inductive-recursive types in [14]. A symmetric scheme for both inductive and coinductive types is given in [5]. Basold et al. [6] define an external syntactic scheme for higher inductive types with only 0-constructors and compute the types of elimination principles. In [27] a semantics is given for the same class of HITs but with no recursive equality constructors. Dybjer and Moeneclaey define a syntactic scheme for finitary HITs and show their existence in a groupoid model [15].

Internal codes for simple inductive types such as natural numbers, lists or binary trees can be given by containers which are decoded to W-types [1]. Morris and Altenkirch [22] extend the notion of container to that of indexed container which specifies indexed inductive types. Codes for inductive-recursive types are given in [16]. Inductive-inductive types were introduced by Forsberg together with an internal coding scheme [23]. Sojakova [26] defines a subset of HITs called W-suspensions by an internal coding scheme similar to W-types. She proves that the induction principle is equivalent to homotopy initiality.

Quotient types [17] are precursors of higher inductive types (HITs). The notion of HIT first appeared in [25], however only through examples and without a general definition. Lumsdaine and Shulman give a general specification of models of type theory supporting higher inductive types [21]. They introduce the notion of cell monad with parameters and characterise the class of models which have intial algebras for a cell monad with parameters. Kraus [19] and Van Doorn [12] construct propositional truncation as a sequential colimit. The schemes mentioned so far do not support higher inductive-inductive types.

The closest to our work is the article of Altenkirch et al. [2] which gives a categorical specification of quotient inductive-inductive types (QIITs), i.e. set-truncated higher inductive-inductive types. Sorts are specified as a list of functors into Set where the domain of the functor is a category constructed from results of the previous functors, thus encoding dependencies of later sorts on previous ones. The constructors are specified mutually with their category of algebras and underlying carrier functor. The specification supports set-level equality constructors. From a specification of a QIIT they derive the type of the eliminator and show that this corresponds to initiality.

The logical predicate syntactic translation was introduced by Bernardy et al. [8]. The idea that a context can be seen as a definition of an inductive type and the logical predicate translation can be used to derive the types of motives and methods was described in [3, Section 5.3]. Logical relations are used to derive the computation rules in [18, Section 4.3], however only for closed QIITs. Syntactic translations in the context of the calculus of inductive constructions are discussed in [10]. Logical relations and parametricity can also be used to justify the *existence* of inductive types in a type theory with an impredicative universe [4]. In contrast, we only use logical relations to *describe* HIITs.

2 Target type theory

In this section we describe the target type theory. It is the target of our translations, and it also serves as a source of constants which are external to the HIIT being defined. It has Russell-style universes, Π, Σ, equality (identity) and unit type. Our notation is close to Agda's: we use named variables, terms are identified up to α-conversion, substitution and weakening are implicit. To distinguish notation from the theory of codes described in the next section, we write term formers and metavariables in brick red colour. We have the following judgement kinds.

$\vdash \Gamma$ $\qquad$ Γ is a valid target context

$\Gamma \vdash t : A$ $\qquad$ the target term t has target type A in target context Γ

We only describe the target type theory in informal English instead of writing down all the rules, since they are standard. See [25, Appendix A.2] for a formal treatment.

Context extension is written $\Gamma, x : A$. We have a cumulative hierarchy of universes Type_i.

Dependent function space is denoted $(x : A) \to B$. We write $A \to B$ if B does not depend on x, and $\to$ associates to the right, $(x : A)(y : B) \to C$ abbreviates $(x : A) \to (y : B) \to C$ and $(x\, y : A) \to B$ abbreviates $(x : A)(y : A) \to B$. We write $\lambda x.t$ for abstraction and $t\, u$ for left-associative application.

$(x : A) \times B$ stands for Σ types, $A \times B$ for the non-dependent version and $\times$ associates to the left. The constructor for Σ types is denoted (t, u) with eliminators proj_1 and proj_2. Both Π and Σ have definitional β and η rules.

The equality (identity) type for a type A and elements $t : A$, $u : A$ is denoted $t =_A u$ and comes with the constructor refl_t and eliminator J with definitional β-rule. The notation is $\mathsf{J}_{A\, t\, P}\, pr\, {}_u\, eq$ for $t : A$, $P : (x : A) \to t =_A x \to \mathsf{Type}_i$, $pr : P\, t\, \mathsf{refl}$ and $eq : t =_A u$. Sometimes we omit parameters in subscripts.

We will use the following functions defined using J in the standard way. We write $\mathsf{tr}_P\, e\, t : P\, v$ for transport of $t : P\, u$ along $e : u = v$. We write $\mathsf{ap}\, f\, e : f\, u = f\, v$ for $f : A \to B$ and $e : u = v$, $\mathsf{apd}\, f\, e : \mathsf{tr}_P\, e\, (f\, u) = f\, v$ for $f : (x : A) \to B$ and $e : u = v$.

The unit type is denoted $\top$ with constructor tt.

3 Source type theory

The source type theory is the target type theory extended with the following judgement kinds.

$\Gamma \vdash \Delta$ $\qquad$ Δ is a context in the target context Γ

$\Gamma; \Delta \vdash A$ $\qquad$ A is a type in context Δ and target context Γ

$\Gamma; \Delta \vdash t : A$ $\qquad$ t is a term of type A in context Δ and target context Γ

(1) Contexts and variables

$$\frac{\vdash \Gamma}{\Gamma \vdash \cdot} \qquad \frac{\Gamma; \Delta \vdash A}{\Gamma \vdash \Delta, x : A} \qquad \frac{\Gamma; \Delta \vdash A}{\Gamma; \Delta, x : A \vdash x : A} \qquad \frac{\Gamma; \Delta \vdash x : A \qquad \Gamma; \Delta \vdash B}{\Gamma; \Delta, y : B \vdash x : A}$$

(2) Universe

$$\frac{\Gamma; \vdash \Delta}{\Gamma; \Delta \vdash \mathsf{U}} \qquad \frac{\Gamma; \Delta \vdash a : \mathsf{U}}{\Gamma; \Delta \vdash \underline{a}}$$

(3) Inductive parameters

$$\frac{\Gamma; \Delta \vdash a : \mathsf{U} \qquad \Gamma; \Delta, x : \underline{a} \vdash B}{\Gamma; \Delta \vdash (x : a) \to B} \qquad \frac{\Gamma; \Delta \vdash t : (x : a) \to B \qquad \Gamma; \Delta \vdash u : \underline{a}}{\Gamma; \Delta \vdash t\, u : B[x \mapsto u]}$$

(4) Equality

$$\frac{\Gamma; \Delta \vdash a : \mathsf{U} \qquad \Gamma; \Delta \vdash t : \underline{a} \qquad \Gamma; \Delta \vdash u : \underline{a}}{\Gamma; \Delta \vdash t =_a u : \mathsf{U}} \qquad \frac{\Gamma; \Delta \vdash t : \underline{a}}{\Gamma; \Delta \vdash \mathsf{refl} : \underline{t =_a t}}$$

$$\frac{\begin{array}{l} \Gamma; \Delta \vdash t : \underline{a} \\ \Gamma; \Delta, x : \underline{a}, z : \underline{t =_a x} \vdash p : \mathsf{U} \\ \Gamma; \Delta \vdash pr : \underline{p[x \mapsto t, z \mapsto \mathsf{refl}]} \\ \Gamma; \Delta \vdash u : \underline{a} \\ \Gamma; \Delta \vdash eq : \underline{t =_a u} \end{array}}{\Gamma; \Delta \vdash \mathsf{J}_{a\, t\, (x.z.p)}\, pr\, _u\, eq : \underline{p[x \mapsto u, z \mapsto eq]}}$$

$$\frac{\Gamma; \Delta \vdash t : \underline{a} \qquad \Gamma; \Delta, x : \underline{a}, z : \underline{t =_a x} \vdash p : \mathsf{U} \qquad \Gamma; \Delta \vdash pr : \underline{p[x \mapsto t, z \mapsto \mathsf{refl}]}}{\Gamma; \Delta \vdash \mathsf{J}\beta_{a\, t\, (x.z.p)}\, pr : \underline{(\mathsf{J}_{a\, t\, (x.z.p)}\, pr\, _t\, \mathsf{refl}) =_{p[x \mapsto t, z \mapsto \mathsf{refl}]}\, pr}}$$

(5) Non-inductive parameters

$$\frac{\Gamma \vdash A : \mathsf{Type}_0 \qquad \Gamma; \vdash \Delta \qquad (\Gamma, x : A); \Delta \vdash B}{\Gamma; \Delta \vdash (x : A) \to B}$$

$$\frac{\Gamma; \Delta \vdash t : (x : A) \to B \qquad \Gamma \vdash u : A}{\Gamma; \Delta \vdash t\, u : B[x \mapsto u]}$$

(6) Infinitary parameters

$$\frac{\Gamma \vdash A : \mathsf{Type}_0 \qquad \Gamma; \vdash \Delta \qquad (\Gamma, x : A); \Delta \vdash b : \mathsf{U}}{\Gamma; \Delta \vdash (x : A) \to b : \mathsf{U}}$$

$$\frac{\Gamma; \Delta \vdash t : \underline{(x : A) \to b} \qquad \Gamma \vdash u : A}{\Gamma; \Delta \vdash t\, u : \underline{b[x \mapsto u]}}$$

■ **Figure 1** The theory of HIIT codes (part of the source type theory). Substitution and weakening are implicit, we assume fresh names everywhere and consider α-convertible terms equal. The $\Gamma;$ assumptions are not used or changed in parts (1)–(4).

We name the subset of rules of the source theory which derives these judgements *the theory of codes*. The derivation rules are listed in figure 1. A context Δ for which $\Gamma \vdash \Delta$ can be derived represents a specification of a HIIT.

Although every judgement is valid up to a context in the target type theory, note that none of the rules in (1)–(4) depend on or change these assumptions, so they can be safely ignored until part (5). We explain the rules in order.

(1) The rules for context formation and variables are standard. We assume fresh names everywhere to avoid name capture. Note that weakening is implicit.

(2) There is a universe U, with decoding written as an underline (usually El in the literature). Type formation rules will target U. With this part of the syntax we can already define contexts specifying the empty type, unit type and booleans:

$$\cdot,\ Empty : \mathsf{U} \qquad \cdot,\ Unit : \mathsf{U},\ tt : \underline{Unit} \qquad \cdot,\ Bool : \mathsf{U},\ true : \underline{Bool},\ false : \underline{Bool}$$

(3) We have a function space with small domain and large codomain. This can be used to add inductive arguments to type formation rules and constructors. As U is not closed under this function space, these function types cannot (recursively) appear in inductive arguments, which ensures strict positivity. When the codomain does not depend on the domain, $a \to B$ can be written instead of $(x : a) \to B$.

Now we can specify the natural numbers as a context:

$$\cdot,\ Nat : \mathsf{U},\ zero : \underline{Nat},\ suc : Nat \to \underline{Nat}$$

We can also encode inductive-inductive definitions such as the fragment of the well-typed syntax of a type theory mentioned in the introduction:

$$\cdot,\ Con : \mathsf{U},\ Ty : Con \to \mathsf{U},\ \bullet : \underline{Con},\ -\triangleright- : (\Delta : Con) \to Ty\,\Delta \to \underline{Con},$$
$$U : (\Delta : Con) \to \underline{Ty\,\Delta},\ \Pi : (\Delta : Con)(A : Ty\,\Delta)(B : Ty\,(\Delta \triangleright A)) \to \underline{Ty\,\Delta}$$

(4) U is closed under the equality type, with eliminator J and a weak (non-definitional) β-rule. Weakness is required because the syntactic translation $-^{\mathsf{E}}$ defined in Section 4.3 does not preserve this β-rule strictly. Adding equality to the theory of codes allows higher constructors and inductive equality parameters as well. We can now define the circle HIT as the following context:

$$\cdot,\ S^1 : \mathsf{U},\ base : \underline{S^1},\ loop : \underline{base =_{S^1} base}$$

The J rule allows constructors to mention operations on paths as well. For instance, the definition of the torus depends on path composition, which can be defined using J: given $p : \underline{t =_a u}$ and $q : \underline{u =_a v}$, $p \bullet q$ abbreviates $\mathsf{J}_{a\,u\,x.z.(t=x)}\,p\,_v\,q : \underline{t =_a v}$. The torus is given as follows.

$$\cdot,\ T^2 : \mathsf{U},\ b : \underline{T^2},\ p : \underline{b =_{T^2} b},\ q : \underline{b =_{T^2} b},\ t : \underline{p \bullet q =_{(b=_{T^2}b)} q \bullet p}$$

With the equality type at hand, we can define a full well-typed syntax of type theory as given e.g. in [3] as an inductive type (see the examples in the formalisation described in Section 6).

So far we were only able to define closed HIITs, which excludes lists of a given type or the integers as given in the introduction. This is where we need the target theory to be included in the source theory. A context Δ for which $\Gamma \vdash \Delta$ holds can be seen as a specification of an inductive type which depends on Γ. In the case of lists, Γ will be $A : \mathsf{Type}_0$. In the case of integers, we need $Nat : \mathsf{Type}_0$ and $-+- : Nat \to Nat \to Nat$ from Γ.

(5) We have a function space where the domain is a type in the target theory. We distinguish it from (3) by using red brick : instead of : in the domain specification. We specify lists and the integers as follows.

$$A : \mathsf{Type}_0 \vdash \cdot,\ List : \mathsf{U},\ \ nil : \underline{List},\ \ cons : (x : A) \to List \to \underline{List}$$

$$\Gamma \qquad \vdash \cdot,\ Int : \mathsf{U},\ \ pair : (x\,y : Nat) \to \underline{Int},$$

$$eq : (a\,b\,c\,d : Nat)(p : a{+}d =_{Nat} b{+}c) \to \underline{pair\,a\,b =_{Int} \mathsf{pair}\,c\,d},$$

$$trunc : (xy : Int)(p\,q : a =_{Int} b) \to \underline{p =_{x=_{Int}y} q}$$

In the case of integers, Γ is $Nat : \mathsf{Type}_0, -+- : Nat \to Nat \to Nat$, or alternatively, we could require natural numbers in the target theory. As another example, propositional truncation for a type A is specified as follows.

$$A : \mathsf{Type}_0 \vdash \cdot,\ \ tr : \mathsf{U},\ \ emb : (x : A) \to \underline{tr},\ \ eq : (x\,y : tr) \to \underline{x =_{tr} y}$$

The smallness of A is required in (5). It is possible to generalize the syntax of HIITs to arbitrary universe levels, but it is not essential to the current development. Note that the (5) function space preserves strict positivity, since in the target theory there is no way to recursively refer to the inductive type *being defined*. The situation is analogous to the case of W-types [1], where shapes and positions can contain arbitrary types but they cannot recursively refer to the W-type being defined.

(6) U is also closed under a function space where the domain is a target theory type and the codomain is a small source theory type. We overload the application notation for non-inductive parameters, as it is usually clear from context which application is meant. The rules allow types with infinitary constructors, for example trees containing A-elements at the leaves and branching by B (which could be an infinite type):

$$A : \mathsf{Type}_0, B : \mathsf{Type}_0 \vdash \cdot,\ \ T : \mathsf{U},\ \ leaf : (x : A) \to \underline{T},\ \ node : ((x : B) \to T) \to \underline{T}$$

Here, $leaf$ has a function type (5) and $node$ has a function type (3) with a function type (6) in the domain. More generally, we can define W-types [1] as follows. S describes the "shapes" of the constructors and P the "positions" where recursive arguments can appear.

$$S : \mathsf{Type}_0, P : S \to \mathsf{Type}_0 \vdash \cdot,\ \ W : \mathsf{U},\ \ sup : (s : S) \to ((p : P\,s) \to W) \to \underline{W}$$

For a more complex infinitary example, see the definition of Cauchy reals in [25, Definition 11.3.2]. It can be also found as an example file in our Haskell implementation.

The invalid examples Ival and Neg cannot be encoded by the theory of codes. For Ival, we can go as far as

$$\cdot,\ Ival : \mathsf{U},\ \ a : \underline{Ival},\ \ b : \underline{Ival},\ \ \sigma : \underline{? =_{Ival} ?}.$$

The first argument of the function $f : (X : \mathsf{Type}) \to X \to X$ is a target theory type, but we only have $Ival : \mathsf{U}$ in the theory of codes. Neg cannot be typed because the first parameter of the constructor con is a function from a small type to a target theory type, and no such functions can be formed.

4   A syntactic translation from the source to the target theory

An inductive type is specified by a context in the theory of codes defined in the previous section. In this section we define the $-^\mathsf{C}$, $-^\mathsf{M}$ and $-^\mathsf{E}$ operations, which work as follows on the code for natural numbers.

$$(\cdot,\, Nat : \mathsf{U},\, zero : \underline{Nat},\, suc : Nat \to \underline{Nat})^{\mathsf{C}}$$
$$\equiv \top \times (n : \mathsf{Type}_0) \times (z : n) \times (n \to n)$$
$$(\cdot,\, Nat : \mathsf{U},\, zero : \underline{Nat},\, suc : Nat \to \underline{Nat})^{\mathsf{M}} (\mathsf{tt}, n, z, s)$$
$$\equiv \top \times (n^M : n \to \mathsf{Type}_i) \times (z^M : n^M\, z) \times \big((x : n) \to n^M\, x \to n^M\, (s\, x)\big)$$
$$(\cdot, Nat : \mathsf{U}, zero : \underline{Nat}, suc : Nat \to \underline{Nat})^{\mathsf{E}} (\mathsf{tt}, n, z, s)\,(\mathsf{tt}, n^M, z^M, s^M)$$
$$\equiv \top \times \big(n^E : (x : n) \to n^M\, x\big) \times (z^E : n^E\, z = z^M) \times \big((x : n) \to n^E\, (s\, x) = n^M\, x\, (n^E\, x)\big)$$

The brick red coloured result of $-^{\mathsf{C}}$ gives the types of the type formation rule and constructors as an iterated Σ-type. Assuming the existence of constructors, $-^{\mathsf{M}}$ returns the types of motives and methods. Assuming the existence of constructors and the corresponding motives and methods, $-^{\mathsf{E}}$ returns the types of the eliminator and computation rules as target theory equalities $=$. The notation above denotes *left*-nested iterated Σ-types. A more precise presentation would replace each variable with a projection from the preceding Σ-type. We use this notation in order to reduce visual clutter.

Each context entry in the theory of codes specifies a type formation rule or a constructor. In general, the last component of a type in a context entry is of three possible forms: it is either U, $\underline{a}$ for some neutral a, or $\underline{t =_a u}$. The following table summarizes the results of the various translations in the mentioned three cases:

return type	$-^{\mathsf{C}}$	$-^{\mathsf{M}}$	$-^{\mathsf{E}}$
U	type formation rule	motive	eliminator
$\underline{a}$	point constructor	method	computation rule
$\underline{t =_a u}$	path constructor	method expressing preservation of equality	higher computation rule

Note that there is no syntactic distinction between the three kinds of constructors above. Any number of them can be introduced in any order, and each constructor can refer to any previous one. A distinguishing feature of our approach is the utilisation of universes instead of structural rules to introduce new sorts and to ensure strict positivity.

The $-^{\mathsf{C}}$, $-^{\mathsf{M}}$ and $-^{\mathsf{E}}$ operations are defined by induction on the derivations of the source syntax. The operations are identity on derivations of target contexts and target terms (of the forms $\vdash \Gamma$ and $\Gamma \vdash t : A$) and derive target terms from theory of codes contexts, types and terms (of the forms $\Gamma \vdash \Delta$, $\Gamma; \Delta \vdash A$ and $\Gamma; \Delta \vdash t : A$, respectively). We only present the non-identity parts with pattern matching notation, describing how a context, type or a term in the theory of codes is converted to a term in the target theory.

All three operations respect definitional equality. This amounts to preserving equalities of the substitution calculus, as there are no β-rules introduced in the theory of codes.

4.1 Type formation rules and constructors

Given a context in the theory of codes, $-^{\mathsf{C}}$ returns the types of type formation rules and constructors as an iterated Σ-type in the target theory. It is specified as follows.

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta^{\mathsf{C}} : \mathsf{Type}_1} \qquad \frac{\Gamma; \Delta \vdash A}{\Gamma \vdash A^{\mathsf{C}} : \Delta^{\mathsf{C}} \to \mathsf{Type}_1} \qquad \frac{\Gamma; \Delta \vdash t : A}{\Gamma \vdash t^{\mathsf{C}} : (\gamma : \Delta^{\mathsf{C}}) \to A^{\mathsf{C}}\, \gamma}$$

Given a context depending on the target context Γ, $-^{\mathsf{C}}$ returns a type in Γ. Given a type in context Δ it returns a family over Δ^{C}. A term is interpreted by a dependent function in the

target theory. The implementation of $-^\mathsf{C}$ is essentially the standard model, where everything is interpreted by its brick red counterpart, except for contexts which are interpreted as iterated Σ-types.

$$
\begin{aligned}
\cdot^\mathsf{C} &:\equiv \top \\
(\Delta, x : A)^\mathsf{C} &:\equiv (\gamma : \Delta^\mathsf{C}) \times A^\mathsf{C}\,\gamma \\
x^\mathsf{C}\,\gamma &:\equiv x^{\text{th}} \text{ component in } \gamma \\
\mathsf{U}^\mathsf{C}\,\gamma &:\equiv \mathsf{Type}_0 \\
(\underline{a})^\mathsf{C}\,\gamma &:\equiv a^\mathsf{C}\,\gamma \\
((x : a) \to B)^\mathsf{C}\,\gamma &:\equiv (x : a^\mathsf{C}\,\gamma) \to B^\mathsf{C}\,(\gamma, x) \\
(t\,u)^\mathsf{C}\,\gamma &:\equiv (t^\mathsf{C}\,\gamma)\,(u^\mathsf{C}\,\gamma) \\
((x : A) \to B)^\mathsf{C}\,\gamma &:\equiv (x : A) \to B^\mathsf{C}\,\gamma \\
(t\,u)^\mathsf{C}\,\gamma &:\equiv (t^\mathsf{C}\,\gamma)\,u \\
(t =_a u)^\mathsf{C}\,\gamma &:\equiv t^\mathsf{C}\,\gamma = u^\mathsf{C}\,\gamma \\
\mathsf{refl}^\mathsf{C}\,\gamma &:\equiv \mathsf{refl} \\
(\mathsf{J}_{a\,t\,(x.z.p)}\,pr\,_u\,eq)^\mathsf{C}\,\gamma &:\equiv \mathsf{J}_{(a^\mathsf{C}\,\gamma)\,(t^\mathsf{C}\,\gamma)\,(\lambda x\,z.p^\mathsf{C}\,(\gamma, x, z))}\,(pr^\mathsf{C}\,\gamma)\,_{(u^\mathsf{C}\,\gamma)}\,(eq^\mathsf{C}\,\gamma) \\
(\mathsf{J}\beta_{a\,t\,(x.z.p)}\,pr)^\mathsf{C}\,\gamma &:\equiv \mathsf{refl} \\
((x : A) \to b)^\mathsf{C}\,\gamma &:\equiv (x : A) \to b^\mathsf{C}\,\gamma \\
(t\,u)^\mathsf{C}\,\gamma &:\equiv (t^\mathsf{C}\,\gamma)\,u
\end{aligned}
$$

For example, $-^\mathsf{C}$ acts as follows on the topological circle:

$$
(\cdot, S^1 : \mathsf{U}, b : \underline{S^1}, loop : \underline{b = b})^\mathsf{C} \equiv \top \times (S^1 : \mathsf{Type}_0) \times (b : S^1) \times (loop : b = b)
$$

The resulting left-nested Σ type could be written explicitly as below. We shall keep to the more readable notation from now on.

$$
\left(x'' : \left(x' : (x : \top) \times \mathsf{Type}_0\right) \times \mathsf{proj}_2\,x'\right) \times (\mathsf{proj}_2\,x'' = \mathsf{proj}_2\,x'')
$$

4.2   Motives and methods

Given a code for an inductive type and the constructors specified by the code, the operation $-^\mathsf{M}$ returns the induction motives and methods.

$-^\mathsf{M}$ is a variant of the unary logical predicate translation of Bernardy et al. [9]. We fix a level i for the universe we would like to eliminate into. For each context Δ, Δ^M is a predicate over the standard interpretation Δ^C. For a type $\Delta \vdash A$, A^M is a predicate over A^C, which also depends on $\gamma : \Delta^\mathsf{C}$ and a witness of $\Delta^\mathsf{M}\,\gamma$. All of these may refer to a target theory context Γ.

$$
\frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta^\mathsf{M} : \Delta^\mathsf{C} \to \mathsf{Type}_{i+1}} \qquad \frac{\Gamma; \Delta \vdash A}{\Gamma \vdash A^\mathsf{M} : (\gamma : \Delta^\mathsf{C}) \to \Delta^\mathsf{M}\,\gamma \to A^\mathsf{C}\,\gamma \to \mathsf{Type}_{i+1}}
$$

For a term t, t^M witnesses that the predicate corresponding to its type holds for t^C. This is often called a *fundamental theorem* in the literature on logical predicates.

$$
\frac{\Gamma; \Delta \vdash t : A}{\Gamma \vdash t^\mathsf{M} : (\gamma : \Delta^\mathsf{C})(\gamma^M : \Delta^\mathsf{M}\,\gamma) \to A^\mathsf{M}\,\gamma\,\gamma^M\,(t^\mathsf{C}\,\gamma)}
$$

We introduce the following shorthand: $t\,\gamma\,\gamma^M$ is abbreviated as $t\,\gamma^2$ for some t expression. The implementation of $-^{\mathsf{M}}$ is given below.

$$
\begin{aligned}
&.^{\mathsf{M}}\,\gamma && :\equiv \top\\
&(\Delta,\,x:A)^{\mathsf{M}}\,(\gamma,\,t) && :\equiv (\gamma^M : \Delta^{\mathsf{M}}\,\gamma) \times A^{\mathsf{M}}\,\gamma^2\,t\\
&x^{\mathsf{M}}\,\gamma^2 && :\equiv x^{\text{th}}\text{ component in }\gamma^M\\
&\mathsf{U}^{\mathsf{M}}\,\gamma^2\,A && :\equiv A \to \mathsf{Type}_i\\
&(\underline{a})^{\mathsf{M}}\,\gamma^2\,t && :\equiv a^{\mathsf{M}}\,\gamma^2\,t\\
&((x:a)\to B)^{\mathsf{M}}\,\gamma^2\,f && :\equiv (x:a^{\mathsf{C}}\,\gamma)(x^M : a^{\mathsf{M}}\,\gamma^2\,x) \to B^{\mathsf{M}}\,(\gamma,x)\,(\gamma^M,x^M)\,(f\,x)\\
&(t\,u)^{\mathsf{M}}\,\gamma^2 && :\equiv (t^{\mathsf{M}}\,\gamma^2)\,(u^{\mathsf{C}}\,\gamma)\,(u^{\mathsf{M}}\,\gamma^2)\\
&((x:A)\to B)^{\mathsf{M}}\,\gamma^2\,f && :\equiv (x:A) \to B^{\mathsf{M}}\,\gamma^2\,(f\,x)\\
&(t\,u)^{\mathsf{M}}\,\gamma^2 && :\equiv t^{\mathsf{M}}\,\gamma^2\,u\\
&(t =_a u)^{\mathsf{M}}\,\gamma^2\,e && :\equiv \mathsf{tr}_{(a^{\mathsf{M}}\,\gamma^2)}\,e\,(t^{\mathsf{M}}\,\gamma^2) = u^{\mathsf{M}}\gamma^2\\
&(\mathsf{refl}_t)^{\mathsf{M}}\,\gamma^2 && :\equiv \mathsf{refl}_{(t^{\mathsf{M}}\,\gamma^2)}\\
&(\mathsf{J}_{a\,t\,(x.z.p)}\,pr\,_u\,eq)^{\mathsf{M}}\,\gamma^2 && :\equiv \mathsf{J}\left(\mathsf{J}\,(pr^{\mathsf{M}}\,\gamma^2)\,(eq^{\mathsf{C}}\,\gamma)\right)(eq^{\mathsf{M}}\,\gamma^2)\\
&(\mathsf{J}\beta_{a\,t\,(x.z.p)}\,pr)^{\mathsf{M}}\,\gamma^2 && :\equiv \mathsf{refl}\\
&((x:A)\to b)^{\mathsf{M}}\,\gamma^2\,f && :\equiv (x:A) \to b^{\mathsf{M}}\,\gamma^2\,(f\,x)\\
&(t\,u)^{\mathsf{M}}\,\gamma^2 && :\equiv t^{\mathsf{M}}\,\gamma^2\,u
\end{aligned}
$$

The predicate for a context is given by iterating $-^{\mathsf{M}}$ for its constituent types. For a variable, the corresponding witness is looked up from γ^M.

The predicate for the universe, given an element of $A : \mathsf{U}^{\mathsf{C}}\,\gamma$ (with $\mathsf{U}^{\mathsf{C}}\,\gamma \equiv \mathsf{Type}_0$) returns the predicate space over A. The predicate for a type $\underline{a}$ is given by the predicate for a.

The predicate for a function type with small domain expresses preservation of predicates (at the domain and codomain types). Witnesses of application are given by recursive application of $-^{\mathsf{M}}$. The definitions for the other (non-inductive) function spaces are similar, except there is no predicate for the domain types, and thus no witnesses are required.

The predicate for the equality type $t =_a u$, for each $e : (t =_a u)^{\mathsf{C}}\,\gamma$, i.e. $e : t^{\mathsf{C}}\,\gamma = u^{\mathsf{C}}\,\gamma$, says that t^{M} and u^{M} are equal. As these have different types, we have to transport over the original equality e. The witness for refl is reflexivity in the target theory. The interpretation of J is given by a double J application, which definition is sourced from [20]. Here, we use a shortened J notation; we refer to the formalisation (Section 6) for the details.

Again, let us consider the circle example:

$$
\begin{aligned}
&(\cdot,\,S^1 : \mathsf{U},\,b : \underline{S^1},\,loop : \underline{b = b})^{\mathsf{M}}\,(\mathsf{tt},\,S^1,\,b,\,loop)\\
&\equiv \top \times (S^{1M} : S^1 \to \mathsf{Type}_i) \times (b^M : S^{1M}\,b) \times (loop^M : \mathsf{tr}_{S^{1M}}\,loop\,b^M = b^M)
\end{aligned}
$$

The inputs of $-^{\mathsf{M}}$ here are the code for the circle (the context in black) and a tuple of the type formation rule S^1, the constructor b and the equality constructor $loop$. It returns a family over the type S^1, an element of this family b^M at index b, and and an equality between b^M and b^M which lies over $loop$.

4.3 Eliminators and computation rules

The operation $-^{\mathsf{E}}$ yields eliminators and computation rules. It is a generalised binary logical relation translation where the type of the second parameter of the relation may depend on the first parameter.

Contexts are interpreted as dependent binary relations between constructors and methods. The universe level i was previously chosen for the $-^{\mathsf{M}}$ operation.

$$\frac{\Gamma \vdash \Delta}{\Gamma ; \Delta^{\mathsf{E}} : (\gamma : \Delta^{\mathsf{C}}) \to \Delta^{\mathsf{M}} \gamma \to \mathsf{Type}_i}$$

Types are interpreted as dependent binary relations which additionally depend on $(\gamma, \gamma^M, \gamma^E)$ interpretations of the context.

$$\frac{\Gamma ; \Delta \vdash A}{\Gamma \vdash A^{\mathsf{E}} : (\gamma : \Delta^{\mathsf{C}})(\gamma^M : \Delta^{\mathsf{M}} \gamma)(\gamma^E : \Delta^{\mathsf{E}} \gamma \gamma^M)(x : A^{\mathsf{C}} \gamma) \to A^{\mathsf{M}} \gamma \gamma^M x \to \mathsf{Type}_i}$$

For a term t, t^{E} witnesses that the relation corresponding to its type holds for t^{C} and t^{M}.

$$\frac{\Gamma ; \Delta \vdash t : A}{\Gamma \vdash t^{\mathsf{E}} : (\gamma : \Delta^{\mathsf{C}})(\gamma^M : \Delta^{\mathsf{M}} \gamma)(\gamma^E : \Delta^{\mathsf{E}} \gamma \gamma^M) \to A^{\mathsf{E}} \gamma \gamma^M \gamma^E (t^{\mathsf{C}} \gamma)(t^{\mathsf{M}} \gamma \gamma^M)}$$

In addition to γ^2, we use $t \gamma^3$ to abbreviate $t \gamma \gamma^M \gamma^E$. The implementation is the following.

$$
\begin{aligned}
\cdot^{\mathsf{E}} \gamma \gamma^M &:\equiv \top \\
(\Delta, x : A)^{\mathsf{E}} (\gamma, t)(\gamma^M, t^M) &:\equiv (\gamma^E : \Delta^{\mathsf{E}} \gamma^2) \times A^{\mathsf{E}} \gamma^3 \, t \, t^M \\
x^{\mathsf{E}} \gamma^3 &:\equiv x^{\text{th}} \text{ component in } \gamma^E \\
\mathsf{U}^{\mathsf{E}} \gamma^3 A \, A^M &:\equiv (x : A) \to A^M \, x \\
(\underline{a})^{\mathsf{E}} \gamma^3 \, t \, t^M &:\equiv a^{\mathsf{E}} \gamma^3 \, t = t^M \\
((x : a) \to B)^{\mathsf{E}} \gamma^3 \, f \, f^M &:\equiv (x : a^{\mathsf{C}} \gamma) \to B^{\mathsf{E}} (\gamma, x)(\gamma^M, a^{\mathsf{E}} \gamma^3 x)(\gamma^E, \mathsf{refl}) \\
&\qquad\qquad\qquad (f \, x)\left(f^M \, x \, (a^{\mathsf{E}} \gamma^3 x)\right) \\
(t \, u)^{\mathsf{E}} \gamma^3 &:\equiv \mathsf{J}\,(t^{\mathsf{E}} \gamma^3 (u^{\mathsf{C}} \gamma))(u^{\mathsf{E}} \gamma^3) \\
((x : A) \to B)^{\mathsf{E}} \gamma^3 \, f \, f^M &:\equiv (x : A) \to B^{\mathsf{E}} \gamma^3 (f \, x)(f^M \, x) \\
(t \, u)^{\mathsf{E}} \gamma^3 &:\equiv t^{\mathsf{E}} \gamma^3 \, u \\
(t =_a u)^{\mathsf{E}} \gamma^3 \, e &:\equiv \mathsf{tr}\,(t^{\mathsf{E}} \gamma^3)\left(\mathsf{tr}\,(u^{\mathsf{E}} \gamma^3)\,(\mathsf{apd}\,(a^{\mathsf{E}} \gamma^3)\,e)\right) \\
(\mathsf{refl}_t)^{\mathsf{E}} \gamma^3 &:\equiv \mathsf{J}\,\mathsf{refl}\,(t^{\mathsf{E}} \gamma^3) \\
(\mathsf{J}_{a\,t\,(x.z.p)}\,pr\,_u\,eq)^{\mathsf{E}} \gamma^3 &:\equiv
\end{aligned}
$$

$$
\mathsf{J}\left(\mathsf{J}\left(\mathsf{J}\left(\mathsf{J}\,(\lambda p^M \, p^E \, pr^M \, pr^E . pr^E)\,(t^{\mathsf{E}} \gamma^3)\right.\right.\right.
$$
$$
\left.\left.\left. (\mathsf{uncurry}\,p^M \gamma^2)\,(\mathsf{uncurry}\,p^E \gamma^3)\,(pr^M \gamma^2)\,(pr^E \gamma^3)\right)\,eq^{\mathsf{C}} \gamma\right)\,u^{\mathsf{E}} \gamma^3\right)(eq^{\mathsf{E}} \gamma^3)
$$

$$
\begin{aligned}
(\mathsf{J}\beta_{a\,t\,(x.z.p)}\,pr)^{\mathsf{E}} \gamma^3 &:\equiv \\
\mathsf{J}\left(\mathsf{J}\,(\lambda p^M \, p^E . \mathsf{refl})\,(t^{\mathsf{E}} \gamma^3)\,(\mathsf{uncurry}\,p^M \gamma^2)\,(\mathsf{uncurry}\,p^E \gamma^3)\right)&(pr^{\mathsf{E}} \gamma^3) \\
((x : A) \to b)^{\mathsf{E}} \gamma^3 \, f \, t &:\equiv b^{\mathsf{E}} \gamma^3 \, (f \, t) \\
(t \, u)^{\mathsf{E}} \gamma^3 &:\equiv \mathsf{ap}\,(\lambda f. f \, u)\,(t^{\mathsf{E}} \gamma^3)
\end{aligned}
$$

The U^{E} and $(\underline{a})^{\mathsf{E}}$ definitions are the key points of the $-^{\mathsf{E}}$ operation. The definitions for the other $-^{\mathsf{E}}$ cases are largely determined by these.

The U^{E} rule yields the type of the eliminator for a type formation rule. In the natural numbers example above, the non-indexed $Nat : \mathsf{U}$ sort is interpreted as $n^E : (x{:}n) \to n^M \, x$. For indexed sorts, the indices are first processed by the $-^{\mathsf{E}}$ cases for inductive and non-inductive parameters, until the ultimate U return type is reached. Hence, the eliminator for a sort is always a function.

Analogously, the $-^\mathsf{E}$ result type for a point or path constructor is always a β-rule, i.e. a function type ending in an equality. To see why, consider the $(\underline{a})^\mathsf{E}$ definition. It expresses that applications of a^E eliminators must be equal to the corresponding t^M induction methods. Hence, for path and point constructor types, $-^\mathsf{E}$ works by first processing all inductive and non-inductive indices, then finally returning an equality type.

Let us also consider the $((x : a) \to B)^\mathsf{E}$ case for inductive parameters. Here, we make crucial use of the fact that the domain type a is small. This provides us $a^\mathsf{E} \gamma^3 x$, which we use to witness the $a^M \gamma^2 x$ hypothesis for B^E. Without the size restriction on inductive parameters (which enforces strict positivity), the $-^\mathsf{E}$ operation would not be possible at all, because a^E for a large a type would be merely an opaque relation instead of an eliminator function.

Here, we only provide abbreviated definitions for the $t\,u$, $t =_a u$, refl, J and Jβ cases. In the J case, we write uncurry p^M for $\lambda \gamma\,\gamma^M\,x\,x^M\,z\,z^M.\,p^M\,(\gamma, x, z)\,(\gamma^M, x^M, z^M)$ and analogously elsewhere. The full definitions can be found in the Agda formalisation. The definitions are highly constrained by the required types, and not particularly difficult to implement with the help of a proof assistant; they all involve doing successive path induction on all equalities available from induction hypotheses, with appropriately generalized induction motives.

The full $(\mathsf{J}_{a\,t\,(x.z.p)}\,pr\,_u\,eq)^\mathsf{E}$ definition is quite large, and, for instance, yields a very large β-rule for the higher inductive torus definition (the reader can confirm this using the Haskell implementation). Nevertheless, an implementation focused on practicality may provide smaller specialized $-^\mathsf{E}$ definitions for commonly used equality operations such as composition or inverses.

The circle example is a bit more interesting here:

$$(\cdot,\, S^1 : \mathsf{U},\, b : \underline{S^1},\, loop : \underline{b = b})^\mathsf{E}\,(\mathsf{tt},\, S^1,\, b,\, loop)\,(\mathsf{tt},\, S^{1M},\, b^M,\, loop^M)$$
$$\equiv \top \times (S^{1E} : (x : S^1) \to S^{1M}\,x) \times (b^E : S^{1E}\,b = b^M)$$
$$\times\,(loop^E : \mathsf{tr}_{(\lambda x.\mathsf{tr}_{\,S^{1M}}\,loop\,x\,=\,b^M)}\,b^E\,(\mathsf{tr}_{(\lambda x.\mathsf{tr}_{\,S^{1M}}\,loop\,(S^{1E}\,b)\,=\,x)}\,b^E\,(\mathsf{apd}\,S^{1E}\,loop))$$
$$=\,loop^M)$$

In homotopy type theory, the β-rule for $loop$ is usually just $\mathsf{apd}\,S^{1E}\,loop = loop^M$, but here all β-rules are propositional, so we need to transport with b^E to make the equation well-typed. When computing the type of $loop^E$, we start with $(\underline{b = b})^\mathsf{E}\,\gamma^3\,loop\,loop^M$. Next, this evaluates to $(b = b)^\mathsf{E}\,\gamma^3\,loop = loop^M$, and then we unfold the left hand side to get the doubly-transported expression in the result.

In Appendix A, we show how the elimination principle is computed for the two-dimensional sphere.

For another example for the translations, we consider indexed W-types which can describe a large class of inductive definitions [22]. Suppose we are in the target context $I : \mathsf{Type}_0, S : \mathsf{Type}_0, P : S \to \mathsf{Type}_0, out : S \to I, in : (s : S) \to P\,s \to I$. Then, the code for the corresponding indexed W-type is the following:

$$W :\equiv (\cdot,\ w : (i : I) \to \mathsf{U},\ \ sup : (s : S) \to ((p : P\,s) \to w\,(in\,s\,p)) \to \underline{w\,(out\,s)})$$

We pick a universe level j for elimination. The interpretations of W are the following,

omitting leading $\top$ components:

$$W^{\mathsf{C}} \equiv (w : I \to \mathsf{Type}_0) \times ((s : S) \to ((p : P\,s) \to w\,(in\,s\,p)) \to w\,(out\,s))$$

$$W^{\mathsf{M}}\,(w, sup) \equiv (w^M : (i : I) \to w\,i \to \mathsf{Type}_j)$$
$$\times \big((s : S)(f : (p : P\,s) \to w\,(in\,s\,p))$$
$$\to ((p : P\,s) \to w^M\,(in\,s\,p)\,(f\,p)) \to w^M\,(out\,s)\,(sup\,s\,f)\big)$$

$$W^{\mathsf{E}}\,(w, sup)\,(w^M, sup^M) \equiv$$
$$(w^E : (i : I)(x : w\,i) \to w^M\,i\,x)$$
$$\times \big((s : S)(f : (p : P\,s) \to w\,(in\,s\,p))$$
$$\to w^E\,(out\,s)\,(sup\,s\,f) = sup^M\,s\,f\,(\lambda p.\,w^E\,(in\,s\,p)\,(f\,p))\big)$$

5   Existence of HIITs

The target type theory supports HIITs if whenever we can derive $\Gamma \vdash \Delta$ in the source theory, the following rules are admissible.

$$\frac{}{\Gamma \vdash \mathsf{con}_\Delta : \Delta^{\mathsf{C}}} \qquad\qquad \frac{\Gamma \vdash m : \Delta^{\mathsf{M}}\,\mathsf{con}_\Delta}{\Gamma \vdash \mathsf{elim}_\Delta\,m : \Delta^{\mathsf{E}}\,\mathsf{con}_\Delta\,m}$$

We can add HIITs to the target theory by extending it with the theory of codes (making the target and the source theory the same) and adding the above rules with the additional assumption of $\Gamma \vdash \Delta$. However, this only adds HIITs with weak computation rules. To make the computation rules definitional, we would probably need a two-level target type theory with an equality having an equality reflection rule as in Voevodsky's homotopy type system [28] or Andromeda [7].

6   Formalisation and implementation

There are three additional development artifacts to the current work: a Haskell implementation, a shallow Agda formalisation and a deep Agda formalisation. All three are available from the homepage of the first author.

The Haskell implementation takes as input a file which contains a representation of a $\Gamma \vdash \Delta$ specification of a HIIT. Then, it checks the input with respect to the rules in figure 1, and outputs an Agda-checkable file which contains the results of the $-^{\mathsf{C}}$, $-^{\mathsf{M}}$ and $-^{\mathsf{E}}$ translations. It comes with examples, including the ones in this paper, indexed W-types [22], the dense completion [23, Appendix A.1.3] and several HITs from [25] including the definition of Cauchy reals. It can be checked that our implementation computes the expected elimination principles in these cases.

The shallow Agda formalisation embeds both the source and target theories shallowly into Agda: it represents types as Agda types, functions as Agda functions, and so on. We also leave the $-^{\mathsf{C}}$ operation implicit. We state each case of the $-^{\mathsf{M}}$ and $-^{\mathsf{E}}$ translations as Agda functions from all induction hypotheses to the result type of the translation, which lets us "typecheck" the translation. We have found that this style of formalisation is conveniently light, but remains detailed enough to be useful. We also generated some of the code of the Haskell implementation from this formalisation.

The deep Agda formalisation still uses a shallow embedding of the target type theory, but it uses a deep embedding of the source theory, in the style of [3]. In the formalisation we merge the three translations into a single model construction. This allows us to prove

strict preservation of definitional equalities in the substitution calculus of the source theory, in contrast to the shallow formalisation, where we cannot reason about definitional equalities of Agda terms. Due to technical challenges, this formalisation uses transport instead of J in the source theory, but this still covers a rather large class of HIIT definitions.

7   Conclusions and further work

Higher inductive-inductive types are useful in defining the well-typed syntax of type theory in an abstract way [3]. From a universal algebra point of view, they provide initial algebras for multi-sorted algebraic theories where the sorts can depend on each other. From the perspective of homotopy type theory, they provide synthetic versions of homotopy-theoretic constructions such as higher dimensional spheres or cell complexes. So far, no general scheme of HIITs have been proposed. To quote Lumsdaine and Shulman [21]:

> "The constructors of an ordinary inductive type are unrelated to each other. But in a higher inductive type each constructor must be able to refer to the previous ones; specifically, the source and target of a path-constructor generally involve the previous point-constructors. No syntactic scheme has yet been proposed for this that covers all cases of interest while remaining meaningful and consistent."

In this paper we proposed such a syntactic scheme which also includes inductive-inductive types. We tackled the problem of complex dependencies on previous type formation rules and constructors by a well-known method of describing intricate dependencies: the syntax of type theory itself. We had to limit the type formers to only allow strictly positive definitions, but these restrictions are the only things that a type theorist has to learn to understand our codes. Our encoding is also *direct* in the sense that the types of constructors and eliminators are exactly as required and not merely up to isomorphisms.

In this paper we only *specified* HIITs and characterised their induction principles. Proving their *existence* is left as further work. This would likely involve reducing HIITs to basic building blocks such as W-types and quotient types.

The theory of codes was defined as part of the syntax of another type theory, the target theory. An alternative way would be to define the theory of codes internally to a type theoretic metatheory in the style of [3]. However, as described in [3, Section 6], there would be a coherence problem: for the internal syntax to be useful, we need to truncate it to be a set (as the trunc constructor did for Int in the introduction). As the eliminator needs to respect the equality given by trunc, we would only be able to eliminate from the internal type theory into a set. This would preclude the definition of even the $-^C$ operation, which corresponds to the standard model. A possible solution to this problem would be to add all the higher coherence laws to the syntax (e.g. the pentagon law for the composition of substitutions) and then prove that the syntax is a set. For this however, we would probably need a two-level metatheory as in [11].

When working in a metatheory with uniqueness of identity proofs (which implies that all HIITs are in essence QIITs), the theory of codes admits a category model where the interpretation of a context is the category of algebras corresponding to the context. In the future, we would like to prove that these categories have initial algebras given by terms in the theory of codes.

─── **References** ───

1   Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers — constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005. Applied Semantics: Selected Topics.

2   Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, pages 293–310, Cham, 2018. Springer International Publishing.

3   Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In Rastislav Bodik and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 18–29. ACM, 2016. URL: `http://dl.acm.org/citation.cfm?id=2837614`, `doi:10.1145/2837614.2837638`.

4   Robert Atkey, Neil Ghani, and Patricia Johann. A relationally parametric model of dependent type theory. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 503–516. ACM, 2014. URL: `http://dl.acm.org/citation.cfm?id=2535838`, `doi:10.1145/2535838.2535852`.

5   Henning Basold and Herman Geuvers. Type Theory based on Dependent Inductive and Coinductive Types. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Proceedings of LICS '16*, pages 327–336. ACM, 2016. `doi:10.1145/2933575.2934514`.

6   Henning Basold, Herman Geuvers, and Niels van der Weide. Higher inductive types in programming. *Journal of Universal Computer Science*, 23(1):63–88, jan 2017.

7   Andrej Bauer, Gaëtan Gilbert, Philipp Haselwarter, Matija Pretnar, and Christopher A. Stone. Design and implementation of the andromeda proof assistant. In Silvia Ghilezan and Ivetić Jelena, editors, *22nd International Conference on Types for Proofs and Programs, TYPES 2016*. University of Novi Sad, 2016.

8   Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Parametricity and dependent types. In *ACM Sigplan Notices*, volume 45, pages 345–356. ACM, 2010.

9   Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Proofs for free — parametricity for dependent types. *Journal of Functional Programming*, 22(02):107–152, 2012. `doi:10.1017/S0956796812000056`.

10   Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. The next 700 syntactical models of type theory. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 182–194, New York, NY, USA, 2017. ACM. `doi:10.1145/3018610.3018620`.

11   Paolo Capriotti and Nicolai Kraus. Univalent higher categories via complete semi-segal types. *Proc. ACM Program. Lang.*, 2(POPL):44:1–44:29, dec 2017. `doi:10.1145/3158132`.

12   Floris van Doorn. Constructing the propositional truncation using non-recursive hits. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, pages 122–129, New York, NY, USA, 2016. ACM. `doi:10.1145/2854065.2854076`.

13   Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6:440–465, 1997.

14   Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65:525–549, 2000.

15   Peter Dybjer and Hugo Moeneclaey. Finitary higher inductive types in the groupoid model. *Electronic Notes in Theoretical Computer Science*, 336:119–134, 2018. The Thirty-third Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII). `doi:10.1016/j.entcs.2018.03.019`.

16   Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions. In *Typed Lambda Calculi and Applications, volume 1581 of Lecture Notes in Computer Science*, pages 129–146. Springer, 1999.

17   Martin Hofmann. *Extensional concepts in intensional type theory.* Thesis. University of Edinburgh, Department of Computer Science, 1995.

18   Ambrus Kaposi. *Type theory in a type theory with quotient inductive types.* PhD thesis, University of Nottingham, 2017.

19   Nicolai Kraus. Constructions with non-recursive higher inductive types. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 595–604, New York, NY, USA, 2016. ACM. `doi:10.1145/2933575.2933586`.

20   Marc Lasson. Canonicity of weak ω-groupoid laws using parametricity theory. *Electronic Notes in Theoretical Computer Science*, 308:229–244, 2014. `doi:10.1016/j.entcs.2014.10.013`.

21   Peter LeFanu Lumsdaine and Mike Shulman. Semantics of higher inductive types, 2017. `arXiv:1705.07088`.

22   Peter Morris and Thorsten Altenkirch. Indexed containers. In *Twenty-Fourth IEEE Symposium in Logic in Computer Science (LICS 2009)*, 2009.

23   Fredrik Nordvall Forsberg. *Inductive-inductive definitions.* PhD thesis, Swansea University, 2013.

24   Christine Paulin-Mohring. Inductive definitions in the system Coq — rules and properties. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 1993. `doi:10.1007/BFb0037116`.

25   The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. Technical report, Institute for Advanced Study, 2013.

26   Kristina Sojakova. Higher inductive types as homotopy-initial algebras. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 31–42, New York, NY, USA, 2015. ACM. `doi:10.1145/2676726.2676983`.

27   Niels van der Weide. Higher inductive types. Master's thesis, Radboud University, Nijmegen, 2016.

28   Vladimir Voevodsky. A simple type system with two identity types. Unpublished note, 2013.

A   Elimination principle computed for the two-dimensional sphere

The two-dimensional sphere is given by the following context in the theory of codes.

$$\Gamma :\equiv \left(\cdot, \ \ S^2 : \mathsf{U}, \ \ b : \underline{S^2}, \ \ surf : \underline{\mathsf{refl}_b =_{(b=_{S^2} b)} \mathsf{refl}_b}\right)$$

The sphere-algebras are computed as follows.

$$\Gamma^{\mathsf{C}} \equiv \top \times (S^2 : \mathsf{Type}_0) \times (b : S^2) \times (surf : \mathsf{refl}_b =_{(b=_{S^2} b)} \mathsf{refl}_b)$$

Given a sphere-algebra and fixing a universe level i, the motives and methods are computed as follows.

$$\Gamma^{\mathsf{M}} \ (\mathsf{tt}, \ S^2, \ b, \ surf)$$
$$\equiv \top \times (S^{2M} : S^2 \to \mathsf{Type}_i)$$
$$\times (b^M : S^{2M} \ b)$$
$$\times (surf^M : \mathsf{tr}_{(\mathsf{tr}_{S^{2M}} \ - \ b^M = b^M)} \ surf \ \mathsf{refl}_{b^M} = \mathsf{refl}_{b^M})$$

Given a sphere-algebra and the motives and methods for this algebra, the types of the elimination principles are computed as follows.

$$\Gamma^{\mathsf{E}} \ (\mathsf{tt}, \ S^2, \ b, \ surf) \ (\mathsf{tt}, \ S^{2M}, \ b^M, \ surf^M)$$
$$\equiv \top \times (S^{2E} : (x : S^2) \to S^{2M} \ x)$$
$$\times (b^E : S^{2E} \ b = b^M)$$
$$\times \left(surf^E : \mathsf{tr} \left(\mathsf{J} \, \mathsf{refl} \, b^E \right) \left(\mathsf{tr} \left(\mathsf{J} \, \mathsf{refl} \, b^E \right) \left(\mathsf{apd} \left(\lambda x. \mathsf{tr} \, b^E \left(\mathsf{tr} \, b^E \left(\mathsf{apd} \, S^{2E} \, x \right) \right) \right) surf \right) \right) \right.$$
$$\left. = surf^M \right)$$

Note that if b^E is a definitional equality (that is, we have $S^{2E} \ b \equiv b^M$), the occurrences of b^E in the type of $surf^E$ can be replaced by refl. In this case the type of $surf^E$ becomes the expected $\mathsf{apd} \left(\mathsf{apd} \, S^{2E} \right) surf = surf^M$.

Lifting Coalgebra Modalities and IMELL Model Structure to Eilenberg-Moore Categories

Jean-Simon Pacaud Lemay

University of Oxford, Computer Science Department, Oxford, UK
https://www.cs.ox.ac.uk/people/jean-simon.lemay/
jean-simon.lemay@kellogg.ox.ac.uk
https://orcid.org/0000-0003-4124-3722

── Abstract ──────────────────────────────────

A categorical model of the multiplicative and exponential fragments of intuitionistic linear logic (IMELL), known as a *linear category*, is a symmetric monoidal closed category with a monoidal coalgebra modality (also known as a linear exponential comonad). Inspired by R. Blute and P. Scott's work on categories of modules of Hopf algebras as models of linear logic, we study Eilenberg-Moore categories of monads as models of IMELL. We define an IMELL lifting monad on a linear category as a Hopf monad – in the Bruguieres, Lack, and Virelizier sense – with a mixed distributive law over the monoidal coalgebra modality. As our main result, we show that the linear category structure lifts to Eilenberg-Moore categories of IMELL lifting monads. We explain how monoids in the Eilenberg-Moore category of the monoidal coalgebra modality can induce IMELL lifting monads and provide sources for such monoids. Along the way, we also define mixed distributive laws of bimonads over coalgebra modalities and lifting differential category structure to Eilenberg-Moore categories of exponential lifting monads.

1 Introduction

Linear logic, as introduced by Girard [11], is a resource sensitive logic which due to its flexibility admits multiple different fragments and a wide range of applications. A categorical model of the multiplicative fragment of intuitionistic linear logic (IMILL) [2, 12] is a symmetric monoidal closed category, while a categorical model of IMILL with negation is a ∗-autonomous category [23]. Categories of modules of (cocommutative) Hopf algebras (over a commutative ring) are important and of interest, especially in representation theory, due in part as they are (symmetric) monoidal closed categories [7, 15]. Blute [5] and Scott [4] studied the idea of interpreting categories of modules of Hopf algebras as models of IMILL with negation and its non-commutative variant. If one were instead to look in a more general setting, categories of modules of cocommutative Hopf monoids in arbitrary symmetric monoidal closed categories are again symmetric monoidal closed categories and therefore models of IMILL. But for what

kind of monoids in categorical models of the multiplicative and exponential fragments of intuitionistic linear logic (IMELL), is their categories of modules again a categorical model of IMELL?

The exponential fragment of IMELL adds in the exponential modality which is a unary connective ! — read as either "of course" or "bang" — admitting four structural rules [2, 19]: promotion, dereliction, contraction, and weakening. In terms of the categorical semantics: the exponential modality ! is interpreted as a monoidal coalgebra modality [3] (see Definition 14 below) – also known as a linear exponential comonad [22] – which in particular is a symmetric monoidal comonad (capturing the promotion and dereliction rules) such that for each object A, the exponential $!A$ comes equipped with a natural cocommutative comonoid structure (capturing the contraction and weakening rules). Categorical models of IMELL are known as linear categories [2, 17, 19], which are symmetric monoidal closed categories with monoidal coalgebra modalities.

We can now restate the question we aim to answer in this paper:

- **Question 1:** "For what kind of monoid A in a linear category, is the category of modules of A also a linear category?"

We have already discussed part of the answer regarding the symmetric monoidal closed structure of a linear category: the monoid A needs to be a cocommutative Hopf monoid. What remains to be answered is how to 'extend', or better yet 'lift', the monoidal coalgebra modality to the category of modules of A. Observing that if A is a monoid, then the endofunctor $A \otimes -$ is a monad (see Section 9) and so the category of modules of A corresponds precisely to the category of Eilenberg-Moore algebras of the monad $A \otimes -$. Therefore, we can further generalize the question we want answered:

- **Question 2:** "For what kind of monad on a linear category, is the Eilenberg-Moore category of algebras of that monad also a linear category?"

This now becomes a question of how to lift a comonad to the Eilenberg-Moore category of a monad. And the answer to this question brings us into the realm of distributive laws [1, 26].

Main Definitions and Results

The two main definitions of this paper are exponential lifting monads (Definition 17) and IMELL lifting monads (Definition 20). Briefly, an exponential lifting monad is a symmetric bimonad with a mixed distributive law over a monoidal coalgebra modality, while an IMELL lifting monad is an exponential lifting monad on a linear category which is also a Hopf monad. Proposition 16 provides a partial answer to the Question 2, while Theorem 21 provides the full answer. We summarize these two main results as follows:

- The Eilenberg-Moore category of an exponential lifting monad admits a monoidal coalgebra modality (Proposition 16).
- The Eilenberg-Moore category of an IMELL lifting monad is a linear category (Theorem 21).

Section 9 and Theorem 24 are dedicated to answering Question 1. Summarizing, where recall that for a monoid A, the category of modules over A can be seen as the Eilenberg-Moore category of the monad $A \otimes -$, we have the following two results:

- Monoids in the Eilenberg-Moore category of monoidal coalgebra modalities induce exponential lifting monads (Theorem 23).
- Monoids with antipodes in the Eilenberg-Moore category of monoidal coalgebra modalities of a linear category induce IMELL lifting monads (Theorem 24).

In the process of constructing and defining mixed distributive laws involving monoidal coalgebra modalities, we also discuss mixed distributive laws over the strictly weaker notion

of coalgebra modalities and define coalgebra modality lifting monads (see Definition 12 below) in order to also discuss lifting differential category structure (see Section 11).

Conventions: In these notes, we will use diagrammatic order for composition: this means that the composite map $f; g$ is the map which first does f then g. Also, to simplify working in a symmetric monoidal category, we will instead work in a strict symmetric monoidal category, that is, we will suppress the unit and associativity isomorphisms. For a symmetric monoidal category we use $\otimes$ for the tensor product, K for the monoidal unit, and $\sigma : A \otimes B \to B \otimes A$ for the symmetry isomorphism.

2 Mixed Distributive Laws Between Monads and Comonads

Distributive laws between monads, which are natural transformations satisfying certain coherences with the monad structures, were introduced by Beck [1] in order to both compose monads and lift one monad to the other's Eilenberg-Moore category. By lifting we mean that the forgetful functor from the Eilenberg-Moore category to base category preserves the monad strictly. In fact, there is a bijective correspondence between distributive laws between monads and lifting of monads. From a higher category theory perspective, a distributive law is a monad on the 2-category of monads of a 2-category [24]. There are also several other notions of distributive laws involving monads and bijective correspondence with certain liftings [26]. Of particular interest for this paper are mixed distributive laws of monads over comonads [1] (see Definition 3 below). For a more detailed introduction on distributive laws and liftings see [26].

If only to introduce notation, we first quickly review the notions of monads and their algebras, and the dual notions of comonads and their coalgebras [14, Chapter VI].

▶ **Definition 1.** A **monad** on a category $\mathbb{X}$ is a triple (T, μ, η) consisting of a functor $\mathsf{T} : \mathbb{X} \to \mathbb{X}$ and two natural transformations $\mu : \mathsf{TT}A \to \mathsf{T}A$ and $\eta : A \to \mathsf{T}A$ such that:

$$\mu; \mu = \mathsf{T}(\mu); \mu \qquad \eta; \mu = 1 = \mathsf{T}(\eta); \mu \tag{1}$$

A T**-algebra** for a monad (T, μ, η) is a pair (A, ν) consisting of an object A and a map $\nu : \mathsf{T}A \to A$ such that:

$$\mu; \nu = \mathsf{T}(\mu); \nu \qquad \eta; \nu = 1 \tag{2}$$

A T**-algebra morphism** $f : (A, \nu) \to (B, \omega)$ is a map $f : A \to B$ such that $\nu; f = \mathsf{T}(f); \omega$.

The category of T-algebras and T-algebra morphisms is called the **Eilenberg-Moore category of the monad** (T, μ, η) and is denoted $\mathbb{X}^{\mathsf{T}}$. There is a forgetful functor $\mathsf{U}^{\mathsf{T}} : \mathbb{X}^{\mathsf{T}} \to \mathbb{X}$, which is defined on objects as $\mathsf{U}^{\mathsf{T}}(A, \nu) = A$ and on maps as $\mathsf{U}^{\mathsf{T}}(f) = f$.

▶ **Definition 2.** Dually, a **comonad** on a category $\mathbb{X}$ is a triple $(!, \delta, \varepsilon)$ consisting of a functor $! : \mathbb{X} \to \mathbb{X}$ and two natural transformations $\delta : !A \to !!A$ and $\varepsilon : !A \to A$ such that the dual equations of a monad (1) hold. A $!$**-coalgebra** for a comonad $(!, \delta, \varepsilon)$ is a pair (A, ω), consisting of an object A and a map $\omega : A \to !A$ such that the dual equalities of (2) hold, while $!$-coalgebra morphisms are the dual analogue of T-algebra morphisms.

The category of $!$-coalgebras and $!$-coalgebra morphisms is called the **Eilenberg-Moore category of the comonad** $(!, \delta, \varepsilon)$ and is denoted $\mathbb{X}^{!}$. There is also a forgetful functor $\mathsf{U}^{!} : \mathbb{X}^{!} \to \mathbb{X}$.

▶ **Definition 3.** Let (T, μ, η) be a monad and $(!, \delta, \varepsilon)$ a comonad on the same category. A **mixed distributive law of** (T, μ, η) **over** $(!, \delta, \varepsilon)$ [26] is a natural transformation $\lambda : \mathsf{T}!A \to$ $!\mathsf{T}A$ such that the following diagrams commute:

$$
\begin{array}{ccc}
\mathsf{T}\mathsf{T}!A \xrightarrow{\mathsf{T}(\lambda)} \mathsf{T}!\mathsf{T}A \xrightarrow{\lambda} !\mathsf{T}\mathsf{T}A & \qquad & !A \xrightarrow{\eta} \mathsf{T}!A \\
\downarrow{\mu} \qquad\qquad\qquad \downarrow{!(\mu)} & & {}_{!(\eta)}\searrow \quad \downarrow{\lambda} \\
\mathsf{T}!A \xrightarrow{\qquad\lambda\qquad} !\mathsf{T}A & & !\mathsf{T}A
\end{array}
\tag{3}
$$

$$
\begin{array}{ccc}
\mathsf{T}!A \xrightarrow{\mathsf{T}(\delta)} \mathsf{T}!!A \xrightarrow{\lambda} !\mathsf{T}!A & \qquad & \mathsf{T}!A \xrightarrow{\lambda} !\mathsf{T}A \\
\downarrow{\lambda} \qquad\qquad\qquad \downarrow{!(\lambda)} & & {}_{\mathsf{T}(\varepsilon)}\searrow \quad \downarrow{\varepsilon} \\
!\mathsf{T}A \xrightarrow{\qquad\delta\qquad} !!\mathsf{T}A & & \mathsf{T}A
\end{array}
\tag{4}
$$

As with distributive laws between monads, mixed distributive laws allow one to lift comonads to Eilenberg-Moore categories of monads and also to lift monads to Eilenberg-Moore categories of comonads, such that the respective forgetful functors preserve the monads or comonads strictly. In fact, mixed distributive laws are in bijective correspondence with these liftings:

▶ **Theorem 4.** *[25, Theorem IV.1] Let (T, μ, η) be a monad and $(!, \delta, \varepsilon)$ be a comonad on the same category $\mathbb{X}$. Then the following are in bijective correspondence:*
1. *Mixed distributive laws of (T, μ, η) over $(!, \delta, \varepsilon)$;*
2. *Liftings of the comonad $(!, \delta, \varepsilon)$ to $\mathbb{X}^\mathsf{T}$, that is, a comonad $(\tilde{!}, \tilde{\delta}, \tilde{\varepsilon})$ on $\mathbb{X}^\mathsf{T}$ such that the forgetful functor U^T preserves the comonad strictly, that is, the following equalities hold:*

$$
!; \mathsf{U}^\mathsf{T} = \mathsf{U}^\mathsf{T}; \tilde{!} \qquad \mathsf{U}^\mathsf{T}(\tilde{\delta}) = \delta \qquad \mathsf{U}^\mathsf{T}(\tilde{\varepsilon}) = \varepsilon
$$

3. *Liftings of the monad (T, μ, η) to $\mathbb{X}^\mathsf{T}$, that is, a monad $(\tilde{\mathsf{T}}, \tilde{\mu}, \tilde{\eta})$ on $\mathbb{X}^!$ such that the forgetful functor $\mathsf{U}^!$ preserves the comonad strictly, that is, the following equalities hold:*

$$
\mathsf{T}; \mathsf{U}^! = \mathsf{U}^!; \tilde{\mathsf{T}} \qquad \mathsf{U}^!(\tilde{\mu}) = \mu \qquad \mathsf{U}^!(\tilde{\eta}) = \eta
$$

We quickly recall part of how to construct liftings from mixed distributive laws (for more details see [26]). Let λ be a mixed distributive law of (T, μ, η) over $(!, \delta, \varepsilon)$. For a T-algebra (A, ν), the pair $(!A, \nu^\sharp)$ is a T-algebra where the map $\nu^\sharp : \mathsf{T}!A \to !A$ is defined as follows:

$$
\nu^\sharp := \mathsf{T}!A \xrightarrow{\quad\lambda\quad} !\mathsf{T}A \xrightarrow{\quad!(\nu)\quad} !A
\tag{5}
$$

Dually, if (A, ω) is a !-coalgebra, then the pair $(\mathsf{T}A, \omega^\flat)$ is a !-coalgebra where the map $\omega^\flat : \mathsf{T}A \to !\mathsf{T}A$ is defined as follows:

$$
\omega^\flat := \mathsf{T}A \xrightarrow{\quad\mathsf{T}(\omega)\quad} \mathsf{T}!A \xrightarrow{\quad\lambda\quad} !\mathsf{T}A
\tag{6}
$$

To see how to construct mixed distributive laws from liftings, see Appendix A.

3   Coalgebra Modalities

Coalgebra modalities were defined by Blute, Cockett, and Seely when they introduced differential categories [6] and are a strictly weaker notion of monoidal coalgebra modalities.

While monoidal coalgebra modalities are much more popular as they give categorical models of IMELL, coalgebra modalities have sufficient structure to axiomatize differentiation. Therefore, we believe it of interest to discuss liftings and mixed distributive laws over coalgebra modalities in order to also discuss lifting differential category structure (see Section 11). Interesting examples of coalgebra modalities which are not monoidal can be found in [9].

▶ **Definition 5.** In a symmetric monoidal category, a **cocommutative comonoid** is a triple (C, Δ, e) consisting of an object C, a map $\Delta : C \to C \otimes C$ called the **comultiplication**, and a map $\mathsf{e} : C \to K$ called the **counit** such that the following diagrams commute:

$$
\begin{array}{ccc}
C \xrightarrow{\ \Delta\ } C \otimes C & & \\
\Big\downarrow{\scriptstyle\Delta} \qquad \Big\downarrow{\scriptstyle\Delta\otimes 1} & & \\
C \otimes C \xrightarrow[1\otimes\Delta]{} C \otimes C \otimes C & &
\end{array}
\qquad
\begin{array}{c}
C \\
\end{array}
\qquad
\begin{array}{c}
C \xrightarrow{\ \Delta\ } C \otimes C \\
\end{array}
\tag{7}
$$

Coalgebra modalities are comonads with the added property that for each object A, the object $!A$ is naturally a cocommutative comonoid.

▶ **Definition 6.** A **coalgebra modality** [6] on a symmetric monoidal category is a quintuple $(!, \delta, \varepsilon, \Delta, e)$ consisting of a comonad $(!, \delta, \varepsilon)$, a natural transformation $\Delta : !A \to !A \otimes !A$, and a natural transformation $\mathsf{e} : !A \to K$ such that for each object A, the triple $(!A, \Delta, \mathsf{e})$ is a cocommutative comonoid and δ is a comonoid morphism, that is, $\delta; \Delta = \Delta; (\delta \otimes \delta)$ and $\delta; \mathsf{e} = \mathsf{e}$.

Requiring that Δ and e be natural transformations is equivalent to asking that for each map $f : A \to B$, the map $!(f) : !A \to !B$ is a comonoid morphism. Every !-coalgebra (A, ω) of a coalgebra modality $(!, \delta, \varepsilon, \Delta, e)$ comes equipped with a cocommutative comonoid structure [19, 22] with comultiplication $\Delta^\omega : A \to A \otimes A$ and counit $\mathsf{e}^\omega : A \to K$ defined as follows:

$$
\Delta^\omega := A \xrightarrow{\ \omega\ } !A \xrightarrow{\ \Delta\ } !A \otimes !A \xrightarrow{\ \varepsilon\otimes\varepsilon\ } A \otimes A \qquad \mathsf{e}^\omega := A \xrightarrow{\ \omega\ } !A \xrightarrow{\ \mathsf{e}\ } K \tag{8}
$$

Notice that since δ is a comonoid morphism, when applying this construction to a cofree !-coalgebra $(!A, \delta)$ we re-obtain Δ and e, that is, $\Delta^\delta = \Delta$ and $\mathsf{e}^\delta = \mathsf{e}$. Furthermore, by naturality of Δ and e, every !-coalgebra morphisms becomes a comonoid morphism on the induced comonoid structures.

4 Symmetric Bimonads and Lifting Symmetric Monoidal Structure

In order to lift coalgebra modalities to an Eilenberg-Moore category of algebras over a monad, we must at least have that said Eilenberg-Moore category be a symmetric monoidal category such that the forgetful functor be a strict monoidal functor. To achieve this, the monad must also be a symmetric comonoidal monad, which we will here call a *symmetric bimonad* following Bruguieres, Lack, and Virelizier's terminology [7] (orginially introduced under the name Hopf monad by Moerdijk [21]). In short, a symmetric bimonad monad (see Defintion 8 below) is a monad whose underlying endofunctor is symmetric comonoidal such that certain extra compatibilities with the monad structure hold. For a higher category theory approach to the subject, we invite the curious reader to see [27].

▶ **Definition 7.** A **symmetric comonoidal endofunctor** – also known as a symmetric opmonoidal endofunctor [16] – on a symmetric monoidal category $\mathbb{X}$ is a triple $(\mathsf{T}, \mathsf{n}_2, \mathsf{n}_1)$

consisting of an endofunctor $\mathsf{T} : \mathbb{X} \to \mathbb{X}$, a natural transformation $\mathsf{n}_2 : \mathsf{T}(A \otimes B) \to \mathsf{T}A \otimes \mathsf{T}B$, and a map $\mathsf{n}_1 : \mathsf{T}K \to K$ such that the following diagrams commute:

$$
\begin{array}{cc}
\mathsf{T}(A \otimes B \otimes C) \xrightarrow{\mathsf{n}_2} \mathsf{T}(A \otimes B) \otimes \mathsf{T}C & \mathsf{T}A \xrightarrow{\mathsf{n}_2} \mathsf{T}A \otimes \mathsf{T}K \quad \mathsf{T}(A \otimes B) \xrightarrow{\mathsf{T}(\sigma)} \mathsf{T}(B \otimes A) \\
\downarrow{\mathsf{n}_2} \qquad\qquad \downarrow{\mathsf{n}_2 \otimes 1} & \downarrow{\mathsf{n}_2} \qquad \downarrow{1 \otimes \mathsf{n}_1} \quad \downarrow{\mathsf{n}_2} \qquad\qquad \downarrow{\mathsf{n}_2} \\
\mathsf{T}A \otimes \mathsf{T}(B \otimes C) \xrightarrow[1 \otimes \mathsf{n}_2]{} \mathsf{T}A \otimes \mathsf{T}B \otimes \mathsf{T}C & \mathsf{T}K \otimes \mathsf{T}A \xrightarrow[\mathsf{n}_1 \otimes 1]{} {!}A \quad \mathsf{T}A \otimes \mathsf{T}B \xrightarrow[\sigma]{} \mathsf{T}B \otimes \mathsf{T}A
\end{array}
\tag{9}
$$

Of particular importance to us is that symmetric comonoidal endofunctors preserves cocommutative comonoids. Indeed, if (C, Δ, e) is a cocommutative comonoid, then the triple $(\mathsf{T}C, \mathsf{T}(\Delta); \mathsf{n}_2, \mathsf{T}(\mathsf{e}); \mathsf{n}_1)$ is a cocommutative comonoid.

▶ **Definition 8.** A **symmetric bimonad** [7] on a symmetric monoidal category is a symmetric comonoidal monad, that is, a quintuple $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ consisting of a monad (T, μ, η) and a symmetric comonoidal endofunctor $(\mathsf{T}, \mathsf{n}_2, \mathsf{n}_1)$ such that the following diagrams commute:

$$
\begin{array}{cccc}
\mathsf{T}\mathsf{T}(A \otimes B) \xrightarrow{\mu} \mathsf{T}(A \otimes B) & A \otimes B \xrightarrow{\eta} \mathsf{T}(A \otimes B) & \mathsf{T}\mathsf{T}K \xrightarrow{\mu} \mathsf{T}K & K \xrightarrow{\eta} \mathsf{T}K \\
\downarrow{\mathsf{T}(\mathsf{n}_2)} \qquad\qquad \downarrow & \searrow{\eta \otimes \eta} \quad \downarrow{\mathsf{n}_2} & \downarrow{\mathsf{T}(\mathsf{n}_1)} \quad \downarrow{\mathsf{n}_1} & \searrow \quad \downarrow{\mathsf{n}_1} \\
\mathsf{T}(\mathsf{T}A \otimes \mathsf{T}B) \qquad \downarrow{\mathsf{n}_2} & \mathsf{T}A \otimes \mathsf{T}B & \mathsf{T}K \xrightarrow[\mathsf{n}_1]{} K & K \\
\downarrow{\mathsf{n}_2} & & & \\
\mathsf{T}\mathsf{T}A \otimes \mathsf{T}\mathsf{T}B \xrightarrow[\mu \otimes \mu]{} \mathsf{T}A \otimes \mathsf{T}B & & &
\end{array}
\tag{10}
$$

One reason for the name bimonad is that bimonoids (the generalization of bialgebras for arbitrary symmetric monoidal categories) give rise to bimonads [7] as we will explain in Section 9.

As previously advertised, the Eilenberg-Moore category of a symmetric bimonad is a symmetric monoidal category. Define a symmetric monoidal structure on $\mathbb{X}^{\mathsf{T}}$ as follows: the monoidal unit is the pair (K, n_1), while for a pair of T-algebras (A, ν) and (B, ν'), their tensor product is defined as the pair $(A \otimes B, \nu \otimes^{\mathsf{T}} \nu')$ where $\nu \otimes^{\mathsf{T}} \nu'$ is defined as follows:

$$
\nu \otimes^{\mathsf{T}} \nu' := \mathsf{T}(A \otimes B) \xrightarrow{\mathsf{n}_2} \mathsf{T}A \otimes \mathsf{T}B \xrightarrow{\nu \otimes \nu'} A \otimes B
\tag{11}
$$

Therefore, the two left most diagrams of (10) are the statement that n_2 is a T-algebra morphism, while the right most diagrams state that (K, n_1) is a T-algebra. In fact, for a monad on a symmetric monoidal category, symmetric bimonad structures on the monad are in bijective correspondence with symmetric monoidal structures on the Eilenberg-Moore category which are strictly preserved by the forgetful functor [26].

5 Lifting Coalgebra Modalities

We now define the notion of mixed distributive laws between symmetric comonoidal endofunctors and coalgebra modalities, in order to lift coalgebra modalities to the Eilenberg-Moore category of symmetric bimonads.

▶ **Definition 9.** Let $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ be a symmetric bimonad and $({!}, \delta, \varepsilon, \Delta, \mathsf{e})$ be a coalgebra modality on the same symmetric monoidal category. A **mixed distributive law of** $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ **over** $({!}, \delta, \varepsilon, \Delta, \mathsf{e})$ is a mixed distributive law λ of (T, μ, η) over $({!}, \delta, \varepsilon)$ such

that λ is a comonoid morphism, that is, the following diagrams commute:

$$
\begin{array}{ccc}
\mathsf{T}!A \xrightarrow{\lambda} !\mathsf{T}A & \qquad & \mathsf{T}!A \xrightarrow{\lambda} !\mathsf{T}A \\
\mathsf{T}(\Delta) \downarrow \qquad \downarrow \Delta & & \mathsf{T}(\mathsf{e}) \downarrow \qquad \downarrow \mathsf{e} \\
\mathsf{T}(!A \otimes !A) & & \mathsf{T}K \xrightarrow{\mathsf{n}_1} K \qquad (12) \\
\mathsf{n}_2 \downarrow & & \\
\mathsf{T}!A \otimes \mathsf{T}!A \xrightarrow{\lambda \otimes \lambda} !\mathsf{T}A \otimes !\mathsf{T}A & &
\end{array}
$$

We first observe that these mixed distributive laws preserve the induced comonoid structure on !-coalgebras in the following sense:

▶ **Lemma 10.** *Let* $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ *be a symmetric bimonad and* $(!, \delta, \varepsilon, \Delta, \mathsf{e})$ *be a coalgebra modality on the same symmetric monoidal category, and let* λ *be a mixed distributive law of* $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ *over* $(!, \delta, \varepsilon, \Delta, \mathsf{e})$. *If* (A, ω) *is a !-coalgebra, then the following diagrams commute:*

$$
\begin{array}{ccc}
\mathsf{T}A \xrightarrow{\mathsf{T}(\Delta^\omega)} \mathsf{T}(A \otimes A) & \qquad & \mathsf{T}A \xrightarrow{\mathsf{T}(\mathsf{e}^\omega)} \mathsf{T}K \\
\searrow_{\Delta^{\omega^\flat}} \qquad \downarrow \mathsf{n}_2 & & \searrow_{\mathsf{e}^{\omega^\flat}} \qquad \downarrow \mathsf{n}_1 \\
\mathsf{T}A \otimes \mathsf{T}A & & K
\end{array}
$$

where $\omega^\flat$ *is defined as in (6), and both* $\Delta^{\omega^\flat}$ *and* $\mathsf{e}^{\omega^\flat}$ *are defined as in (8).*

Proof. See Appendix B. ◀

Now we give the equivalence between liftings and mixed distributive laws of symmetric bimonads over coalgebra modalities.

▶ **Proposition 11.** *Let* $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ *be a symmetric bimonad and* $(!, \delta, \varepsilon, \Delta, \mathsf{e})$ *be a coalgebra modality on the same symmetric monoidal category* $\mathbb{X}$. *Then the following are in bijective correspondence:*

1. *Mixed distributive laws* $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ *over* $(!, \delta, \varepsilon, \Delta, \mathsf{e})$;
2. *Liftings of* $(!, \delta, \varepsilon, \Delta, \mathsf{e})$ *to* $\mathbb{X}^\mathsf{T}$, *that is, a coalgebra modality* $(\tilde{!}, \tilde{\delta}, \tilde{\varepsilon}, \tilde{\Delta}, \tilde{\mathsf{e}})$ *on* $\mathbb{X}^\mathsf{T}$ *which is a lifting of the underlying comonad* $(!, \delta, \varepsilon)$ *to* $\mathbb{X}^\mathsf{T}$ *(in the sense of Theorem 4) such that* $\mathsf{U}^\mathsf{T}(\tilde{\Delta}) = \Delta$ *and* $\mathsf{U}^\mathsf{T}(\tilde{\mathsf{e}}) = \mathsf{e}$.

Proof. See Appendix B. ◀

We give a name to symmetric bimonads with these mixed distributive laws.

▶ **Definition 12.** *Let* $(!, \delta, \varepsilon, \Delta, \mathsf{e})$ *be a coalgebra modality. A* **coalgebra modality lifting monad** *is a sextuple* $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1, \lambda)$ *consisting of a symmetric bimonad* $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ *and a mixed distributive law* λ *of* $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ *over* $(!, \delta, \varepsilon, \Delta, \mathsf{e})$.

Proposition 11 implies that the Eilenberg-Moore category a coalgebra modality lifting monad admits a coalgebra modality which is strictly preserved by the forgetful functor.

6   Monoidal Coalgebra Modalities

Monoidal coalgebra modalities can be described as coalgebra modalities whose underlying comonad is a symmetric monoidal comonad such that Δ and e are compatible with the symmetric monoidal comonad structure. Symmetric monoidal comonads are simply the dual notion of symmetric bimonads (Definition 8) and could therefore be called symmetric bicomonads. However the name symmetric monoidal comonad is used within the linear logic community and therefore we have elected to keep it here. Though it should be noted that the term bicomonad was used by Bruguieres, Lack, and Virelizier [7].

▶ **Definition 13.** A **symmetric monoidal comonad** is a quintuple $(!, \delta, \varepsilon, \mathsf{m}_2, \mathsf{m}_1)$ consisting of a comonad $(!, \delta, \varepsilon)$, a natural transformation $\mathsf{m}_2 : !A \otimes !B \to !(A \otimes B)$, and a map $\mathsf{m}_1 : K \to !K$ such that $(!, \mathsf{m}_2, \mathsf{m}_1)$ is a symmetric monoidal functor, that is, the dual diagrams of (9) commute, and such that δ and ε are monoidal transformations, that is, the dual diagrams of (10) commute.

As this is the dual notion of symmetric bimonads, the Eilenberg-Moore category of a symmetric monoidal comonad is a symmetric monoidal category.

▶ **Definition 14.** A **monoidal coalgebra modality** [3] (also called a **linear exponential modality** [22]) on a symmetric monoidal category is a septuple $(!, \delta, \varepsilon, \Delta, e, \mathsf{m}_2, \mathsf{m}_1)$ such that $(!, \delta, \varepsilon, \mathsf{m}_2, \mathsf{m}_1)$ is a symmetric monoidal comonad and $(!, \delta, \varepsilon, \Delta, e)$ is a coalgebra modality, and such that Δ and e are monoidal transformations, that is, the following diagrams commute:

$$
\begin{array}{ccc}
!A \otimes !B & \xrightarrow{\ \ \mathsf{m}_2\ \ } & !(A \otimes B) \\
\Delta \otimes \Delta \downarrow & & \downarrow \Delta \\
!A \otimes !A \otimes !B \otimes !B & & \\
1 \otimes \sigma \otimes 1 \downarrow & & \downarrow \\
!A \otimes !B \otimes !A \otimes !B \underset{\mathsf{m}_2 \otimes \mathsf{m}_2}{\rightrightarrows} !(A \otimes B) \otimes !(A \otimes B)
\end{array}
\qquad
\begin{array}{c}
!A \otimes !B \xrightarrow{\mathsf{m}_2} !(A \otimes B) \\
\ \ e \otimes e \searrow \quad \downarrow e \\
K
\end{array}
\quad
\begin{array}{c}
K \xrightarrow{\mathsf{m}_1} !K \\
\mathsf{m}_1 \otimes \mathsf{m}_1 \searrow \downarrow \Delta \\
K
\end{array}
\quad
\begin{array}{c}
K \xrightarrow{\mathsf{m}_1} !K \\
\ \| \searrow \downarrow e \\
!K
\end{array}
\quad (13)
$$

and also that Δ and e are !-coalgebra morphisms, that is, the following diagrams commute:

$$
\begin{array}{ccc}
!A & \xrightarrow{\ \ \ \ \delta\ \ \ \ } & !!A \\
\Delta \downarrow & & \downarrow !(\Delta) \\
!A \otimes !A \xrightarrow{\delta \otimes \delta} !!A \otimes !!A \xrightarrow{\mathsf{m}_2} & !(!A \otimes !A)
\end{array}
\qquad
\begin{array}{ccc}
!A & \xrightarrow{\delta} & !!A \\
e \downarrow & & \downarrow !(e) \\
K & \xrightarrow{\mathsf{m}_1} & !(K)
\end{array}
\qquad (14)
$$

Notice that the monoidal coalgebra modality requirement that Δ and e both be monoidal transformations is equivalent to asking that m_2 and m_1 are both comonoid morphisms. Furthermore, if (A, ω) is a !-coalgebra then Δ^ω and e^ω (as defined in (8)) are both !-coalgebra morphisms. This implies that the tensor product of the Eilenberg-Moore category of a monoidal coalgebra modality is in fact a product [22].

The most well known and common examples of monoidal coalgebra modalities are known as **free exponential modalities** [20]. Free exponential modalities can be described as monoidal coalgebra modalities with the added property that $!A$ is the cofree cocommtuative comonoid over A. In this case, !-coalgebras correspond precisely to the cocommutative comonoids of the symmetric monoidal category. Therefore, the Eilenberg-Moore category of a free exponential modality is equivalent to the category of cocommutative comonoids of the base symmetric monoidal category.

7   Lifting Monoidal Coalgebra Modalities

▶ **Definition 15.** Let $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ be a symmetric bimonad and $(!, \delta, \varepsilon, \Delta, \mathsf{e}, \mathsf{m}_2, \mathsf{m}_1)$ be a monoidal coalgebra modality on the same symmetric monoidal category. A **mixed distributive law** $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ **over** $(!, \delta, \varepsilon, \Delta, \mathsf{e}, \mathsf{m}_2, \mathsf{m}_1)$ is a mixed distributive law λ of $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ over $(!, \delta, \varepsilon, \Delta, \mathsf{e})$ such that the following diagrams commute:

$$
\begin{array}{ccc}
\mathsf{T}(!A \otimes !B) \xrightarrow{\mathsf{n}_2} \mathsf{T}!A \otimes \mathsf{T}!B \xrightarrow{\lambda \otimes \lambda} !\mathsf{T}A \otimes !\mathsf{T}B & \quad & \mathsf{T}K \xrightarrow{\quad \mathsf{n}_1 \quad} K \\
\downarrow{\scriptstyle \mathsf{T}(\mathsf{m}_2)} \qquad\qquad\qquad\qquad\qquad\qquad \downarrow{\scriptstyle \mathsf{m}_2} & & \downarrow{\scriptstyle \mathsf{T}(\mathsf{m}_1)} \qquad\qquad\qquad \downarrow{\scriptstyle \mathsf{m}_1} \\
\mathsf{T}!(A \otimes B) \xrightarrow{\lambda} !\mathsf{T}(A \otimes B) \xrightarrow{!(\mathsf{n}_2)} !(\mathsf{T}A \otimes \mathsf{T}B) & & \mathsf{T}!K \xrightarrow{\lambda} !\mathsf{T}K \xrightarrow{!(\mathsf{n}_1)} !K
\end{array} \tag{15}
$$

▶ **Proposition 16.** *Let* $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ *be a symmetric bimonad and* $(!, \delta, \varepsilon, \Delta, \mathsf{e}, \mathsf{m}_2, \mathsf{m}_1)$ *be a monoidal coalgebra modality on the same symmetric monoidal category* $\mathbb{X}$. *Then the following are in bijective correspondence:*

1. *Mixed distributive laws of* $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ *over* $(!, \delta, \varepsilon, \Delta, \mathsf{e}, \mathsf{m}_2, \mathsf{m}_1)$;
2. *Liftings of the monoidal coalgebra modality* $(!, \delta, \varepsilon, \Delta, \mathsf{e}, \mathsf{m}_2, \mathsf{m}_1)$ *to* $\mathbb{X}^{\mathsf{T}}$, *that is, a monoidal coalgebra modality* $(\tilde{!}, \tilde{\delta}, \tilde{\varepsilon}, \tilde{\Delta}, \tilde{\mathsf{e}}, \tilde{\mathsf{m}}_2, \tilde{\mathsf{m}}_1)$ *on* $\mathbb{X}^{\mathsf{T}}$ *which is a lifting of the underlying coalgebra modality* $(!, \delta, \varepsilon, \Delta, \mathsf{e})$ *to* $\mathbb{X}^{\mathsf{T}}$ *(in the sense of Proposition 11) such that* $\mathsf{U}^{\mathsf{T}}(\tilde{\mathsf{m}}_2) = \mathsf{m}_2$ *and* $\mathsf{U}^{\mathsf{T}}(\tilde{\mathsf{m}}_1) = \mathsf{m}_1$.

Proof. See Appendix C.   ◀

As before, we give a name to symmetric bimonads with these mixed distributive laws.

▶ **Definition 17.** Let $(!, \delta, \varepsilon, \Delta, \mathsf{e}, \mathsf{m}_2, \mathsf{m}_1)$ be a monoidal coalgebra modality. An **exponential lifting monad** is a sextuple $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1, \lambda)$ consisting of a symmetric bimonad $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ and a mixed distributive law λ of $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ over $(!, \delta, \varepsilon, \Delta, \mathsf{e}, \mathsf{m}_2, \mathsf{m}_1)$.

Proposition 16 implies that the Eilenberg-Moore category of an exponential lifting monad admits a monoidal coalgebra modality which is strictly preserved by the forgetful functor.

8   Lifting Linear Category Structure

Categorical models of IMELL are known as linear categories:

▶ **Definition 18.** A **linear category** [2] is a symmetric monoidal closed category with a monoidal coalgebra modality.

As linear categories are categorical models of IMELL [2, 17, 19], there is no shortage of examples of linear categories throughout the literature. Hyland and Schalk provide a very nice list of various kinds examples in [13, Section 2.4]. Linear categories whose monoidal coalgebra modality is in fact a free exponential modality are known as **Lafont categories** [19] – we discuss a particular example of a Lafont category at the end of Section 9.

The last piece of the puzzle is being able to lift the monoidal closed structure of a linear category to the Eilenberg-Moore category of our symmetric bimonad in such a way that the forgetful functor preserves the monoidal closed structure strictly. For this we turn to Bruguieres, Lack, and Virelizier's notion of a Hopf monad. Hopf monads were originally introduced by Bruguieres and Virelizier for monoidal categories with duals [8], but the definition of Hopf monads was later extended to arbitrary monoidal categories by the two previous authors and Lack [7]. We choose the later of the two as the definition is somewhat

simpler. The left and right fusion operators of a symmetric bimonad $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ are the natural transformations h_l and h_r defined respectively as follows:

$$\mathsf{h}_l := \; \mathsf{T}(\mathsf{T}A \otimes B) \xrightarrow{\;\mathsf{n}_2\;} \mathsf{T}\mathsf{T}A \otimes \mathsf{T}B \xrightarrow{\;\mu \otimes 1\;} \mathsf{T}A \otimes \mathsf{T}B$$

$$\mathsf{h}_r := \; \mathsf{T}(A \otimes \mathsf{T}B) \xrightarrow{\;\mathsf{n}_2\;} \mathsf{T}A \otimes \mathsf{T}\mathsf{T}B \xrightarrow{\;1 \otimes \mu\;} \mathsf{T}A \otimes \mathsf{T}B$$

Notice that the fusion operators are T-algebra morphisms.

▶ **Definition 19.** A **symmetric Hopf monad** [7] on a symmetric monoidal category is a symmetric bimonad whose fusion operators are natural isomorphisms.

Extending on [7, Theorem 3.6], the Eilenberg-Moore category of a Hopf monad of a symmetric monoidal closed category, is again a symmetric monoidal closed category such that the forgetful functor preserve the symmetric monoidal closed structure strictly.

▶ **Definition 20.** A IMELL **lifting monad** on a linear category with monoidal coalgebra modality $(!, \delta, \varepsilon, \Delta, \mathsf{e}, \mathsf{m}_2, \mathsf{m}_1)$ is an exponential lifting monad $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1, \lambda)$ whose underlying symmetric bimonad is also a symmetric Hopf monad.

▶ Remark. It is worth mentioning that in the definitions of a monoidal coalgebra modality monad and of an IMELL lifting monad, we do not require that the underlying endofunctors of these monads be linearly distributive functors between linear categories in the sense of Hyland and Schalk [13, Definition 4] or that of Melliés [18, Definition 9].

IMELL lifting monads provide us with following main result of this paper:

▶ **Theorem 21.** *The Eilenberg-Moore category of an* IMELL *lifting monad is a linear category such that the forgetful functor preserves the linear category structure strictly.*

9   What Monoids Give IMELL Lifting Monads?

As explained in the introduction, a particular example of Eilenberg-Moore categories we are interested are those arising as categories of modules over monoids. Indeed, endofunctors of the form $A \otimes -$ for some object A, admit a monad structure precisely when the object A is a monoid. Recall that a monoid of a monoidal category is a triple (A, ∇, u) consisting of an object A, a map $\nabla : A \otimes A \to A$ called the multiplication, and a map $\mathsf{u} : K \to A$ called the unit such that the dual of the left and center diagrams of (7) commute (in particular we do not require the multiplication to be commutative). For a monoid (A, ∇, u), the algebras of the monad $(A \otimes -, \nabla \otimes 1, \mathsf{u} \otimes 1)$ are more commonly known as (left) A-modules, and in this case, we denote the Eilenberg-Moore category instead by $\mathsf{MOD}(A)$.

▶ **Definition 22.** In a symmetric monoidal category, a **bimonoid** is a quintuple $(A, \nabla, \mathsf{u}, \Delta, \mathsf{e})$ such that (A, ∇, u) is a monoid, (A, Δ, e) is a comonoid, and the following diagrams commute:

$$(16)$$

A **Hopf monoid** in a symmetric monoidal category is a sextuple $(H, \nabla, \mathsf{u}, \Delta, \mathsf{e}, \mathsf{S})$ consisting of a bimonoid $(H, \nabla, \mathsf{u}, \Delta, \mathsf{e})$ and a map $\mathsf{S} : H \to H$ called the antipode such that the following diagram commutes:

$$
\begin{array}{ccc}
& H \otimes H \xrightarrow{\quad 1 \otimes \mathsf{S} \quad} H \otimes H & \\
\Delta \nearrow & & \searrow \nabla \\
H \xrightarrow{\quad \mathsf{e} \quad} K \xrightarrow{\quad \mathsf{u} \quad} & & \\
\Delta \searrow & & \nearrow \nabla \\
& H \otimes H \xrightarrow{\quad \mathsf{S} \otimes 1 \quad} H \otimes H &
\end{array}
\tag{17}
$$

As previously hinted at, endofunctors of the form $A \otimes -$ admit a symmetric bimonad (resp. symmetric Hopf monad) structure precisely when the object A admits a bimonoid (resp. Hopf monoid) structure whose comultiplication is cocommutative. For details on these constructions see [7, Example 2.10].

Our goal is now to find bimonoids and Hopf monoids which induce exponential lifting monads and IMELL lifting monads. For this we turn to monoids in the Eilenberg-Moore categories of monoidal coalgebra modalities. A monoid in the Eilenberg-Moore category of a monoidal coalgebra modality $(!, \delta, \varepsilon, \Delta, \mathsf{e}, \mathsf{m}_2, \mathsf{m}_1)$ can be seen as a quadruple $(A, \omega, \nabla^\omega, \mathsf{u}^\omega)$ consisting of a !-coalgebra (A, ω) and a monoid $(A, \nabla^\omega, \mathsf{u}^\omega)$ such that ∇^ω and u^ω are !-coalgebra morphisms, that is, the following diagrams commute:

$$
\begin{array}{ccc}
A \otimes A \xrightarrow{\quad \nabla^\omega \quad} A & \qquad & K \xrightarrow{\quad \mathsf{u}^\omega \quad} A \\
{\scriptstyle \omega \otimes \omega} \downarrow \qquad\qquad \downarrow {\scriptstyle \omega} & & {\scriptstyle \mathsf{m}_1} \downarrow \qquad\qquad \downarrow {\scriptstyle \omega} \\
!A \otimes !A \xrightarrow[\mathsf{m}_2]{} !(A \otimes A) \xrightarrow[!(\nabla^\omega)]{} !A & & !K \xrightarrow[!(\mathsf{u}^\omega)]{} !(A)
\end{array}
\tag{18}
$$

However we just mentioned that $A \otimes -$ admit a symmetric bimonad structure if and only if A admits a bimonoid structure with cocommutative comultiplication. Therefore we could instead ask for bimonoids. But it turns out that we only need to ask for monoids instead! To see this, consider monoids in a cartesian monoidal category – which is a category with finite products regarded as a symmetric monoidal category. Every object in a cartesian monoidal category is a cocommutative comonoid and every map is a comonoid morphism. Therefore, since the bimonoid identities are equivalent to requiring that the multiplication and unit be comonoid morphisms, every monoid in a cartesian monoidal category is automatically a cocommutative bimonoid. Since the Eilenberg-Moore category of a monoidal coalgebra modality is a cartesian monoidal category [19, 22], every monoid will be a bimonoid with cocommutative comultiplication.

Following this observation, we obtain the main result of this section:

▶ **Theorem 23.** *Let* $(!, \delta, \varepsilon, \Delta, \mathsf{e}, \mathsf{m}_2, \mathsf{m}_1)$ *be a monoidal coalgebra modality on a symmetric monoidal category* $\mathbb{X}$*. Then the following are in bijective correspondence:*

1. *Monoids in* $\mathbb{X}^!$*;*

2. *Objects* A *such that the endofunctor* $A \otimes -$ *admits an exponential monad lifting structure*

whose mixed distributive law λ satisfies the following:

$$
\begin{array}{ccc}
A \otimes {!}X \otimes {!}Y & \xrightarrow{\ \lambda \otimes 1\ } & {!}(A \otimes X) \otimes {!}Y \\
{\scriptstyle 1 \otimes \mathsf{m}_2}\Big\downarrow & & \Big\downarrow{\scriptstyle \mathsf{m}_2} \\
A \otimes {!}(X \otimes Y) & \xrightarrow{\ \ \lambda\ \ } & {!}(A \otimes X \otimes Y)
\end{array}
\tag{19}
$$

Therefore, each monoid in the Eilenberg-Moore category of a monoidal coalgebra modality induces an exponential lifting monad.

Proof. We only show how to construct one from the other as the proof is somewhat lengthy. Let $(A, \omega, \nabla^\omega, \mathsf{u}^\omega)$ be a monoid in $\mathbb{X}^!$. Define the natural transformation (natural by construction) $\omega^\natural : A \otimes {!}X \to {!}(A \otimes X)$ as follows:

$$
\omega^\natural := \ A \otimes {!}X \xrightarrow{\ \ \omega \otimes 1\ \ } {!}A \otimes {!}X \xrightarrow{\ \ \mathsf{m}_2\ \ } {!}(A \otimes X)
\tag{20}
$$

Then $\omega^\natural$ is a mixed distributive law of the symmetric bimonad structure of $A \otimes -$ over the monoidal coalgebra modality. Conversely, let λ be a mixed distributive law of the exponential lifting monad structure of $A \otimes -$. By applying (6) to the !-coalgebra (K, m_1), we obtain the !-coalgebra $(A, \mathsf{m}_1^\flat)$ where recall:

$$
\mathsf{m}_1^\flat := \ A \xrightarrow{\ \ 1 \otimes \mathsf{m}_1\ \ } A \otimes {!}K \xrightarrow{\ \ \lambda\ \ } {!}A
\tag{21}
$$

Furthermore, the comonoid structure on A induced by the symmetric bimonad structure on $A \otimes -$ is precisely the same as the comonoid structure on A induced by the coalgebra modality from (8). It then follows that the multiplication and unit of A induced by the symmetric bimonad structure on $A \otimes -$ are !-coalgebra morphisms and therefore A admits a monoid structure in $\mathbb{X}^!$. ◀

▶ Remark. It is worth pointing out that commutivity of diagram (19) is only necessary for the bijective correspondence.

As a source of such monoids, since monoidal endofunctors preserve monoids (dual to what was discussed in Section 4), every monoid (A, ∇, u) of the base symmetric monoidal category $\mathbb{X}$ induces a monoid $({!}A, \delta, \nabla^\delta, \mathsf{u}^\delta)$ in $\mathbb{X}^!$ where $\nabla^\delta := \mathsf{m}_2 ; {!}(\nabla)$ and $\mathsf{u}^\delta := \mathsf{m}_1 ; {!}(\mathsf{u})$. In particular, since the monoidal unit K admits a canonical monoid structure, the quadruple $({!}K, \delta, \mathsf{m}_2, \mathsf{m}_1)$ is a monoid in $\mathbb{X}^!$. Another source of such monoids is discussed in the next section.

Recall that in the special case of a free exponential modality, its Eilenberg-Moore category is equivalent to the category of cocommutative comonoids of the base symmetric monoidal category. Therefore, to give a monoid in this Eilenberg-Moore category is precisely to give a bimonoid with cocommutative comultiplication of the base symmetric monoidal category. In fact the category of monoids in the Eilenberg-Moore category of a free exponential modality is equivalent to the category of bimonoids with cocommutative comultiplication of the base category.

For a bimonoid, there is a unique antipode (if it exists) [15] which makes it into a Hopf monoid. Therefore we can easily extended Theorem 23 for IMELL lifting monads.

▶ **Theorem 24.** *Let $({!}, \delta, \varepsilon, \Delta, \mathsf{e}, \mathsf{m}_2, \mathsf{m}_1)$ be a monoidal coalgebra modality of a linear category $\mathbb{X}$. Then the following are in bijective correspondence:*

1. *Monoids $(A, \omega, \nabla^\omega, \mathsf{u}^\omega)$ of $\mathbb{X}^!$ such that there exists an antipode S for the bimonoid $(A, \nabla^\omega, \mathsf{u}^\omega, \Delta^\omega, \mathsf{e}^\omega)$;*

2. *Objects A such that the endofunctor $A \otimes -$ admits an IMELL lifting monad structure whose mixed distributive law satisfies (19).*

Therefore, if A is an object of a linear category which admits a monoid structure with antipode in the Eilenberg-Moore category of the monoidal coalgebra modality, then $\mathsf{MOD}(A)$ is a linear category.

A source of such monoids with antipodes can be found in the next section (Theorem 28).

For Lafont categories – which recall are linear categories whose monoidal coalgebra modality is a free exponential modalitiy – to give a monoid as in Theorem 24 is precisely to give a Hopf monoid with cocommutative comultiplication. As an example, consider the category of vector spaces over a field K, which is a Lafont category where the construction of $!V$, which in this case is known as the cofree cocommutative K-coalgebra over V, can be found here [13, Section 2.4]. Particular examples of Hopf K-algebras [15] with cocommutative comultiplication include the polynomial rings $K[x_1, \ldots, x_n]$, the tensor algebra $\mathsf{T}(V)$ over a K-vector spaces V, the group K-algebra $K[G]$ over an arbitrary group G, and also the field K itself.

10 IMELL Lifting Monads from Additive Structure

We've already seen that monoids in the base category provide a source of monoids in the Eilenberg-Moore category of a monoidal coalgebra modality. In this section we turn to monoidal coalgebra modalities over additive symmetric monoidal categories to provide us with another source of monoids in the Eilenberg-Moore category of said monoidal coalgebra modalities. Here we mean "additive" in the Blute, Cockett, and Seely sense of the term [6], that is, to mean enriched over commutative monoids. In particular, we do not assume negatives (at least not yet...see Theorem 27) nor do we assume biproducts — which differs from other definitions of an additive category found in the literature [14].

▶ **Definition 25.** An **additive category** is a commutative monoid enriched category, that is, a category in which each hom-set is a commutative monoid with an addition operation $+$ and a zero 0, and such that composition preserves the additive structure, that is $k; (f + g); h = k; f; h + k; g; h$ and $0; f = 0 = f; 0$. An **additive symmetric monoidal category** is a commutative monoid enriched symmetric monoidal category, that is, symmetric monoidal category which is also an additive category in which the tensor product is compatible with the additive structure in the sense that $(f + g) \otimes h = f \otimes h + g \otimes h$ and $0 \otimes f = 0$.

It is worth mentioning that every additive category can be completed to a category with biproducts (which is itself an additive category), and similarly every additive symmetric monoidal category can be completed to a additive symmetric monoidal category with biproducts. For this reason, it is possible to argue [10] that one should always assume a setting with biproducts. The problem is that arbitrary coalgebra modalities do not necessarily extend to the biproduct completion. However, monoidal coalgebra modalities induce monoidal coalgebra modalities on the biproduct completion (see [9] for more details).

If $(!, \delta, \varepsilon, \Delta, \mathsf{e}, \mathsf{m}_2, \mathsf{m}_1)$ is a monoidal coalgebra modality on an additive symmetric monoidal category, then $!A$ comes equipped with a monoid structure [9, Theorem 19] where the multiplication ∇ and unit u are both $!$-coalgebra morphisms [9, Lemma 20], [10, Theorem 3.1]. Therefore we obtain the following:

▶ **Lemma 26.** *Every cofree coalgebra of a monoidal coalgebra modality on an additive symmetric monoidal category induces an exponential lifting monad. In particular, for each object A,* MOD$(!A)$ *is an additive symmetric monoidal category with a monoidal coalgebra modality.*

As promised, we will now add negatives to the story of additive symmetric monoidal categories. In particular we will now show that cofree coalgebras of monoidal coalgebra modalities on additive symmetric monoidal categories are Hopf monoids precisely when the additive symmetric monoidal category also admits additive inverses, i.e. negatives. This statement should not be too surprising for two reasons. The first reason is that for a Hopf algebra, the antipode is the bialgebra convolution [15] inverse to the identity. The second reason is that monoidal coalgebra modalities on additive symmetric monoidal categories are strongly connected to the additive structure [9]. A category enriched over abelian groups can be seen as an additive category such that each map f admits an additive inverse, that is, a map $-f$ such that $f + (-f) = 0$. Actually, for an additive category to be enriched over abelian groups, one only requires that the identity maps 1 have additive inverses -1.

▶ **Proposition 27.** *Let $(!, \delta, \varepsilon, \Delta, \mathsf{e}, \mathsf{m}_2, \mathsf{m}_1)$ be a monoidal coalgebra modality on an additive symmetric monoidal category. Then there exists a natural transformation $\mathsf{S} : !A \to !A$ such that for each object A, the septuple $(!A, \nabla, \mathsf{u}, \Delta, \mathsf{e}, \mathsf{S})$ is a cocommutative Hopf monoid (where ∇ and u are defined as in [9]) if and only if the additive symmetric monoidal category is enriched over abelian groups.*

Proof. We only give how to construct antipodes from negatives and conversly negatives from antipodes. Suppose our additive symmetric monoidal category is enriched over abelian groups. Define the antipode $\mathsf{S} : !A \to !A$ as $\mathsf{S} := !(-1)$. Conversely, suppose there exists a natural transformation $\mathsf{S} : !A \to !A$ such that for each object A, the septuple $(!A, \nabla, \mathsf{u}, \Delta, \mathsf{e}, \mathsf{S})$ is a cocommutative Hopf monoid. As previously mentioned, it suffices to give an additive inverse for the identity morphisms. Then for each object A, define the map $-1_A : A \to A$ as follows:

$$-1_A := A \xrightarrow{\ \mathsf{m}_1 \otimes 1\ } !K \otimes A \xrightarrow{\ \mathsf{S} \otimes 1\ } !K \otimes A \xrightarrow{\ \varepsilon \otimes 1\ } A \tag{22}$$

◀

Therefore we obtain the following:

▶ **Theorem 28.** *Every cofree coalgebra of a monoidal coalgebra modality of a linear category which is also an additive symmetric monoidal category enriched over abelian groups, induces an* IMELL *lifting monad. In particular, for each object A,* MOD$(!A)$ *is a linear category which is also an additive symmetric monoidal category enriched over abelian groups.*

11   Lifting Differential Category Structure

In this final section, we briefly recall the notion of differential categories and discuss lifting differential category structure. For more details on differential categories see [3, 6, 9].

▶ **Definition 29.** A differential category [6] is an additive symmetric monoidal category with a coalgebra $(!, \delta, \varepsilon, \Delta, \mathsf{e})$ equipped with a deriving transformation, that is, a natural transformation $\mathsf{d} : !A \otimes A \to !A$ satisfying the identities found in [6, Definition 2.5].

Similar to our discussion on lifting coalgebra modalities, in order to be able to lift differential category structure we will need that the Eilenberg-Moore category of our symmetric bimonad be an additive symmetric monoidal category. In order to achieve this, we will need the underlying endofunctor of our symmetric bimonad to be additive.

▶ **Definition 30.** An **additive functor** between additive categories is a functor which preserves the additive structure strictly, that is, a functor T such that $\mathsf{T}(f + g) = \mathsf{T}(f) + \mathsf{T}(g)$ and $\mathsf{T}(0) = 0$.

One can easily check that for a monad on an additive category whose underlying endofunctor is additive, that its Eilenberg-Moore category is also an additive category such that the forgetful functor preserves the additive structure strictly. Similarly, for a symmetric bimonad on an additive symmetric monoidal category whose underlying endofunctor is additive, its Eilenberg-Moore category is also an additive category such that the forgetful functor preserves the additive symmetric monoidal structure strictly. Luckily for us for any additive symmetric monoidal category, our favourite endofunctor $A \otimes -$ is additive for any object A.

▶ **Definition 31.** Let $\mathbb{X}$ be a differential category with coalgebra modality $(!, \delta, \varepsilon, \Delta, \mathsf{e})$ equipped with deriving transformation d, and let $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ be a symmetric bimonad on $\mathbb{X}$ whose underlying endofunctor is additive. A **mixed distributive law of** $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ **over** $(!, \delta, \varepsilon, \Delta, \mathsf{e})$ **with deriving transformation** d is a mixed distributive law λ of $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ over $(!, \delta, \varepsilon, \Delta, \mathsf{e})$ such that the following diagram commutes:

$$\begin{array}{ccccc}
\mathsf{T}(!A \otimes A) & \xrightarrow{\ \mathsf{n}_2\ } & \mathsf{T}!A \otimes \mathsf{T}A & \xrightarrow{\ \lambda \otimes 1\ } & !\mathsf{T}A \otimes \mathsf{T}A \\
{\scriptstyle \mathsf{T}(\mathsf{d})}\downarrow & & & & \downarrow{\scriptstyle \mathsf{d}} \\
\mathsf{T}!A & & \xrightarrow[\qquad\qquad\lambda\qquad\qquad]{} & & !\mathsf{T}A
\end{array} \tag{23}$$

▶ **Proposition 32.** *Let $\mathbb{X}$ be a differential category with coalgebra modality $(!, \delta, \varepsilon, \Delta, \mathsf{e})$ equipped with deriving transformation d, and let $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ be a symmetric bimonad on $\mathbb{X}$ whose underlying endofunctor is additive. Then the following are in bijective correspondence:*
1. *Mixed distributive laws $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ over $(!, \delta, \varepsilon, \Delta, \mathsf{e})$ with deriving transformation d ;*
2. *Liftings of d to $\mathbb{X}^\mathsf{T}$, that is, a deriving transformation $\tilde{\mathsf{d}}$ for the lifted coalgebra modality $(\tilde{!}, \tilde{\delta}, \tilde{\varepsilon}, \tilde{\Delta}, \tilde{\mathsf{e}})$ on $\mathbb{X}^\mathsf{T}$ from Proposition 11 such that $\mathsf{U}^\mathsf{T}(\tilde{\mathsf{d}}) = \mathsf{d}$.*

Proof. See Appendix D. ◀

In a differential category whose coalgebra modality is also a monoidal coalgebra modality, the deriving transformation and the monoidal coalgebra modality are compatible in the sense of [9, Theorem 25]. And therefore it follows that:

▶ **Theorem 33.** *In a differential category with a monoidal coalgebra modality, the category of modules over a monoid in the Eilenberg-Moore category of the monoidal coalgebra modality is a differential category.*

───── **References** ─────

1   Jon Beck. Distributive laws. In *Seminar on triples and categorical homology theory*, pages 119–140. Springer, 1969.
2   Gavin M Bierman. What is a categorical model of intuitionistic linear logic? In *International Conference on Typed Lambda Calculi and Applications*, pages 78–93. Springer, 1995.

3   R Blute, J Robin B Cockett, and Robert AG Seely. Cartesian differential storage categories. *Theory and Applications of Categories*, 30(18):620–686, 2015.

4   Richard Blute and Philip Scott. Category theory for linear logicians. *Linear logic in computer science*, 316:3–65, 2004.

5   Richard F Blute. Hopf algebras and linear logic. *Mathematical Structures in Computer Science*, 6(2):189–212, 1996.

6   Richard F Blute, J Robin B Cockett, and Robert AG Seely. Differential categories. *Mathematical structures in computer science*, 16(6):1049–1083, 2006.

7   Alain Bruguieres, Steve Lack, and Alexis Virelizier. Hopf monads on monoidal categories. *Advances in Mathematics*, 227(2):745–800, 2011.

8   Alain Bruguieres and Alexis Virelizier. Hopf monads. *Advances in Mathematics*, 215(2):679–733, 2007.

9   J Robin B Cockett and Jean-Simon Lemay. There is only one notion of differentiation. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 84. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

10   M.P. Fiore. Differential structure in models of multiplicative biadditive intuitionistic linear logic. In *International Conference on Typed Lambda Calculi and Applications*, pages 163–177. Springer, 2007.

11   Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987.

12   Jean-Yves Girard and Yves Lafont. Linear logic and lazy computation. In *International Joint Conference on Theory and Practice of Software Development*, pages 52–66. Springer, 1987.

13   Martin Hyland and Andrea Schalk. Glueing and orthogonality for models of linear logic. *Theoretical computer science*, 294(1-2):183–231, 2003.

14   S. Mac Lane. *Categories for the working mathematician*. Springer-Verlag, New York, Berlin, Heidelberg, 1971, revised 2013.

15   Shahn Majid. *Foundations of quantum group theory*. Cambridge university press, 2000.

16   Paddy McCrudden. Opmonoidal monads. *Theory Appl. Categ*, 10(19):469–485, 2002.

17   Paul-André Mellies. Categorical models of linear logic revisited. *Archive ouverte HAL*, 2003. URL: `https://hal.archives-ouvertes.fr/hal-00154229`.

18   Paul-André Mellies. Comparing hierarchies of types in models of linear logic. *Information and Computation*, 189(2):202–234, 2004.

19   Paul-André Mellies. Categorical semantics of linear logic. *Panoramas et syntheses*, 27:15–215, 2009.

20   Paul-André Melliès, Nicolas Tabareau, and Christine Tasson. An explicit formula for the free exponential modality of linear logic. *Mathematical Structures in Computer Science*, pages 1–34, 2017.

21   Ieke Moerdijk. Monads on tensor categories. *Journal of Pure and Applied Algebra*, 168(2):189–208, 2002.

22   Andrea Schalk. What is a categorical model of linear logic. *Manuscript, available from http://www. cs. man. ac. uk/ schalk/work. html*, 2004.

23   Robert AG Seely. *Linear logic,*-autonomous categories and cofree coalgebras*. Ste. Anne de Bellevue, Quebec: CEGEP John Abbott College, 1987.

24   Ross Street. The formal theory of monads. *Journal of Pure and Applied Algebra*, 2(2):149–168, 1972.

25   Donovan H Van Osdol. Sheaves in regular categories. In *Exact Categories and Categories of Sheaves*, pages 223–239. Springer, 1971.

26   Robert Wisbauer. Algebras versus coalgebras. *Applied Categorical Structures*, 16(1):255–295, 2008.

27   Marek Zawadowski. The formal theory of monoidal monads. *Journal of Pure and Applied Algebra*, 216(8-9):1932–1942, 2012.

A   From Liftings to Mixed Distributive Laws

As we will need to know how to construct mixed distributive laws from liftings and vice-versa for the proofs of Propositions 11, 16, and 23, we quickly recall part of these constructions here (for more details see [26]). Constructing liftings from mixed distrubitve laws was discussed at the end of Section 2.

Let $(\tilde{!}, \tilde{\delta}, \tilde{\varepsilon})$ be a lifting of $(!, \delta, \varepsilon)$ to the Eilenberg-Moore category $\mathbb{X}^{\mathsf{T}}$ of a monad (T, μ, η). This implies that for each free T-algebra $(\mathsf{T}A, \mu)$ we have that $\tilde{!}(\mathsf{T}A, \mu) = (!\mathsf{T}A, \mu^\sharp)$ for some map $\mu^\sharp : \mathsf{T}!\mathsf{T}A \to !\mathsf{T}A$. Define the natural transformation $\lambda : \mathsf{T}!A \to !\mathsf{T}A$ as follows:

$$\lambda := \; \mathsf{T}!A \xrightarrow{\;\;\mathsf{T}!(\eta)\;\;} \mathsf{T}!\mathsf{T}A \xrightarrow{\;\;\mu^\sharp\;\;} !\mathsf{T}A \tag{24}$$

Then λ is a mixed distributive law of (T, μ, η) over $(!, \delta, \varepsilon)$.

B   Proofs of Lemma 10 and Proposition 11

Proof of Lemma 10. The lemma follows from commutativity of the following diagrams:

Proof of Proposition 11. The bijective correspondence will follow from Theorem 4. It remains to show that the induced lifting of the comonad from the mixed distributive law is also a lifting of the colgebra modality, and similarly for the mixed distributive law from the lifting of the coalgebra modality.

(1) $\Rightarrow$ (2): Let λ be a mixed distributive of $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ over $(!, \delta, \varepsilon, \Delta, \mathsf{e})$. Consider the induced lifting of $(!, \delta, \varepsilon)$ from Theorem 4. To prove that we have a lifting of the coalgebra modality, it suffices to show that Δ and e are T-algebra morphisms. Then if (A, ν) is a T-algebra, commutativitiy of the following diagrams show that Δ and e are T-algebra

morphisms:

$$
\begin{array}{ccccc}
\mathsf{T}(!A) & \xrightarrow{\ \lambda\ } & !\mathsf{T}(A) & \xrightarrow{\ !(\nu)\ } & !A \\
{\scriptstyle \mathsf{T}(\Delta)}\downarrow & (12) & \downarrow{\scriptstyle \Delta} & \text{Nat. of } \Delta & \downarrow{\scriptstyle \Delta} \\
\mathsf{T}(!A \otimes !A) & \xrightarrow{\ \mathsf{n}_2\ } \mathsf{T}(!A) \otimes \mathsf{T}(!A) \xrightarrow{\ \lambda\otimes\lambda\ } & !\mathsf{T}(A) \otimes !\mathsf{T}(A) & \xrightarrow{\ !(\nu)\otimes!(\nu)\ } & !A \otimes !A
\end{array}
$$

$$
\begin{array}{ccc}
\mathsf{T}(!A) & \xrightarrow{\ \lambda\ } !\mathsf{T}(A) \xrightarrow{\ !(\nu)\ } & !A \\
{\scriptstyle \mathsf{T}(\mathsf{e})}\downarrow \quad (12) \quad {\scriptstyle \mathsf{e}}\downarrow & \text{Nat. of } \mathsf{e} & \\
\mathsf{T}(K) & \xrightarrow{\ \mathsf{n}_1\ } K \xleftarrow{\ \mathsf{e}\ } &
\end{array}
$$

$(2) \Rightarrow (1)$: Let $(\tilde{!}, \tilde{\delta}, \tilde{\varepsilon}, \tilde{\Delta}, \tilde{\mathsf{e}})$ be a lifting of $(!, \delta, \varepsilon, \Delta, \mathsf{e})$ to $\mathbb{X}^{\mathsf{T}}$. This implies that Δ and e are T-algebra morphisms, which in particular for free T-algebras $(\mathsf{T}A, \delta)$, the following diagrams commute:

$$
\begin{array}{ccc}
\mathsf{T}!\mathsf{T}A & \xrightarrow{\ \mu^\sharp\ } & !\mathsf{T}A \\
{\scriptstyle \mathsf{T}(\Delta)}\downarrow & & \downarrow{\scriptstyle \Delta} \\
\mathsf{T}(!A \otimes !A) & \xrightarrow[\mathsf{n}_2]{} \mathsf{T}!\mathsf{T}A \otimes \mathsf{T}!\mathsf{T}A \xrightarrow[\mu^\sharp\otimes\mu^\sharp]{} & !\mathsf{T}A \otimes !\mathsf{T}A
\end{array}
\qquad
\begin{array}{ccc}
\mathsf{T}!\mathsf{T}A & \xrightarrow{\mu^\sharp} & !\mathsf{T}A \\
{\scriptstyle \mathsf{T}(\mathsf{e})}\downarrow & & \downarrow{\scriptstyle \mathsf{e}} \\
\mathsf{T}K & \xrightarrow{\mathsf{n}_1} & K
\end{array}
\quad (25)
$$

where recall $\mu^\sharp$ is the T-algebra structure of $\tilde{!}(\mathsf{T}A, \mu) = (!\mathsf{T}A, \mu^\sharp)$. Consider now the induced mixed distributive law λ of (T, μ, η) over $(!, \delta, \varepsilon)$ as defined in (24). Then that λ is also a mixed distributive law of $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ over $(!, \delta, \varepsilon, \Delta, \mathsf{e})$ follows from commutativity of the following diagrams:

$$
\begin{array}{ccccc}
\mathsf{T}!A & \xrightarrow{\mathsf{T}!(\eta)} & \mathsf{T}!\mathsf{T}A & \xrightarrow{\ \mu^\sharp\ } & !\mathsf{T}A \\
{\scriptstyle \mathsf{T}(\Delta)}\downarrow & \text{Nat. of } \Delta \ {\scriptstyle \mathsf{T}(\Delta)}\downarrow & & & \downarrow{\scriptstyle \Delta} \\
\mathsf{T}(!A \otimes !A) & \xrightarrow[\mathsf{T}(!(\eta)\otimes!(\eta))]{} & \mathsf{T}(!\mathsf{T}A \otimes !\mathsf{T}A) & (25) & \\
{\scriptstyle \mathsf{n}_2}\downarrow & \text{Nat. of } \mathsf{n}_2 \ {\scriptstyle \mathsf{n}_2}\downarrow & & & \downarrow \\
\mathsf{T}!A \otimes \mathsf{T}!A & \xrightarrow[\mathsf{T}!(\eta)\otimes\mathsf{T}!(\eta)]{} & \mathsf{T}!\mathsf{T}A \otimes \mathsf{T}!\mathsf{T}A & \xrightarrow[\mu^\sharp\otimes\mu^\sharp]{} & !\mathsf{T}A \otimes !\mathsf{T}A
\end{array}
$$

$$
\begin{array}{ccc}
\mathsf{T}!A & \xrightarrow{\mathsf{T}!(\eta)} \mathsf{T}!\mathsf{T}A \xrightarrow{\mu^\sharp} & !\mathsf{T}A \\
& \text{Nat. of } \mathsf{e} \ {\scriptstyle \mathsf{T}(\mathsf{e})}\downarrow \ (25) & \downarrow{\scriptstyle \mathsf{e}} \\
{\scriptstyle \mathsf{T}(\mathsf{e})} & \searrow \ \mathsf{T}K \xrightarrow{\mathsf{n}_1} & K
\end{array}
$$

◀

C   Proof of Proposition 16

Proof of Proposition 16. We take the same approach as in the proof of Proposition 11. Again, the bijective correspondence will follow from Theorem 4.

$(1) \Rightarrow (2)$: Let λ be a mixed distributive law of $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ over $(!, \delta, \varepsilon, \Delta, \mathsf{e}, \mathsf{m}_2, \mathsf{m}_1)$. Consider the induced lifting of $(!, \delta, \varepsilon, \Delta, \mathsf{e})$ from Proposition 11. To prove that we have a lifting of the monoidal coalgebra modality, it suffices to show that m_2 and m_1 are T-algebra morphisms. The right diagram of (15) is precisely the statement that m_1 is a T-algebra morphism. Then if (A, ν) and (B, ν') are T-algebras, commutativitiy of the following diagrams

show that m_2 is a T-algebra morphism:

$$
\begin{array}{ccccccc}
\mathsf{T}(!A \otimes !B) & \xrightarrow{\;n_2\;} & \mathsf{T}!A \otimes \mathsf{T}!B & \xrightarrow{\;\lambda \otimes \lambda\;} & !\mathsf{T}A \otimes !\mathsf{T}B & \xrightarrow{\;!(\nu)\otimes!(\nu')\;} & !A \otimes !B \\
\downarrow{\scriptstyle \mathsf{T}(m_2)} & & (15) & & \downarrow{\scriptstyle m_2} & \text{Nat. of } m_2 & \downarrow{\scriptstyle m_2} \\
\mathsf{T}!(A \otimes B) & \xrightarrow{\;\lambda\;} & !\mathsf{T}(A \otimes B) & \xrightarrow{\;!(n_2)\;} & !(\mathsf{T}A \otimes \mathsf{T}B) & \xrightarrow{\;!(\nu\otimes\nu')\;} & !(A \otimes B)
\end{array}
$$

$(2) \Rightarrow (1)$: Let $(\tilde{!}, \tilde{\delta}, \tilde{\varepsilon}, \tilde{\Delta}, \tilde{\mathsf{e}}, \tilde{\mathsf{m}}_2, \tilde{\mathsf{m}}_1)$ be a lifting of $(!, \delta, \varepsilon, \Delta, \mathsf{e}, \mathsf{m}_2, \mathsf{m}_1)$ to $\mathbb{X}^{\mathsf{T}}$. In particular, this implies that m_2 and m_1 are T-algebra morphisms. In particular for free T-algebras $(\mathsf{T}A, \mu)$ and the T-algebra (K, n_1), we have that the following diagrams commute:

$$
\begin{array}{ccccc}
\mathsf{T}(!\mathsf{T}A \otimes !\mathsf{T}B) & \xrightarrow{\;n_2\;} & \mathsf{T}!\mathsf{T}A \otimes !\mathsf{T}!B & \xrightarrow{\;\mu^{\sharp}\otimes\mu^{\sharp}\;} & !\mathsf{T}A \otimes !\mathsf{T}B \\
\downarrow{\scriptstyle \mathsf{T}(m_2)} & & & & \downarrow{\scriptstyle m_2} \\
\mathsf{T}!(\mathsf{T}A \otimes \mathsf{T}B) & & \xrightarrow{\;(\mu\otimes^{\mathsf{T}}\mu)^{\sharp}\;} & & !(\mathsf{T}A \otimes \mathsf{T}B)
\end{array}
\qquad
\begin{array}{ccc}
\mathsf{T}K & \xrightarrow{\;n_1\;} & K \\
\downarrow{\scriptstyle \mathsf{T}(m_1)} & & \downarrow{\scriptstyle m_1} \\
\mathsf{T}!K & \xrightarrow{\;n_1^{\sharp}\;} & !K
\end{array}
\quad (26)
$$

where recall for a T-algebra (A, ν), the map $\nu^{\sharp}$ is the induced T-algebra on $\tilde{!}(A, \nu) = (!A, \nu^{\sharp})$, and $\mu \otimes^{\mathsf{T}} \mu$ is defined as in (11). Notice that since both n_2 and n_1 are T-algebra morphisms, the lifting implies that $!(\mathsf{n}_2)$ and $!(\mathsf{n}_1)$ are also, that is, the following diagrams commute:

$$
\begin{array}{ccc}
\mathsf{T}!\mathsf{T}(A \otimes B) & \xrightarrow{\;\mu^{\sharp}\;} & !\mathsf{T}(A \otimes B) \\
\downarrow{\scriptstyle \mathsf{T}!(n_2)} & & \downarrow{\scriptstyle !(n_2)} \\
\mathsf{T}!(\mathsf{T}A \otimes \mathsf{T}B) & \xrightarrow{\;(\mu\otimes^{\mathsf{T}}\mu)^{\sharp}\;} & !(\mathsf{T}A \otimes \mathsf{T}B)
\end{array}
\qquad
\begin{array}{ccc}
\mathsf{T}!\mathsf{T}K & \xrightarrow{\;\mu^{\sharp}\;} & !\mathsf{T}K \\
\downarrow{\scriptstyle \mathsf{T}!(n_1)} & & \downarrow{\scriptstyle !(n_1)} \\
\mathsf{T}!K & \xrightarrow{\;n_1^{\sharp}\;} & !K
\end{array}
\quad (27)
$$

Consider the induced mixed distributive law λ of $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ over $(!, \delta, \varepsilon, \Delta, \mathsf{e})$ from Proposition 11. Then that λ is a mixed distributive law of $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ over $(!, \delta, \varepsilon, \Delta, \mathsf{e}, \mathsf{m}_2, \mathsf{m}_1)$ follows from commutativity of the following diagrams:

$$
\begin{array}{c}
\mathsf{T}K \xrightarrow{\;\;\mathsf{n}_1\;\;} K
\end{array}
$$

The following commutative diagram appears:

$$
\begin{array}{ccc}
\mathsf{T}K & \xrightarrow{\mathsf{n}_1} & K \\
\mathsf{T}(\mathsf{m}_1)\downarrow & (26) & \downarrow \mathsf{m}_1 \\
& \mathsf{T}!K \xrightarrow{\;\mathsf{n}_1^\sharp\;} & \\
\mathsf{T}!K & (10)\;\; \mathsf{T}!(\mathsf{n}_1)\uparrow\;\;(27) & !K \\
\;\;\xrightarrow{\mathsf{T}!(\eta)} \mathsf{T}!\mathsf{T}K \xrightarrow{\mu^\sharp} !\mathsf{T}K \xrightarrow{!(\mathsf{n}_1)} !K
\end{array}
$$

◀

D   Proof of Proposition 32

Proof Proposition 32. The bijective correspondence will follow immediately from Proposition 11. Therefore, it remains to show that we can obtain one from the other.

$(1) \Rightarrow (2)$: Let λ be a mixed distributive law of $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ over $(!, \delta, \varepsilon, \Delta, \mathsf{e})$ with deriving transformation d. Consider the induced lifting of $(!, \varepsilon, \varepsilon, \Delta, \mathsf{e})$ from Proposition 11. To prove that we have a lifting of the deriving transformation, it suffices to show that d is a T-algebra morphism. Then if (A, ν) is a T-algebra, commutativitiy of the following diagram shows that d is a T-algebra morphism:

$$
\begin{array}{ccccccc}
\mathsf{T}(!A \otimes A) & \xrightarrow{\mathsf{n}_2} & \mathsf{T}!A \otimes \mathsf{T}A & \xrightarrow{\lambda \otimes 1} & !\mathsf{T}A \otimes \mathsf{T}A & \xrightarrow{!(\nu) \otimes \nu} & !A \otimes A \\
\mathsf{T}(\mathsf{d})\downarrow & & (23) & & \downarrow \mathsf{d} & \text{Nat. of. } \mathsf{d} & \downarrow \mathsf{d} \\
\mathsf{T}!A & & \xrightarrow{\hspace{5em}\lambda\hspace{5em}} & & !\mathsf{T}A & \xrightarrow{!(\nu)} & !A
\end{array}
$$

$(2) \Rightarrow (1)$: Let $\tilde{\mathsf{d}}$ be a lifting of d to $\mathbb{X}^\mathsf{T}$. This implies that d is a T-algebra morphism, which in particular for free T-algebras $(\mathsf{T}A, \mu)$, the following diagram commutes:

$$
\begin{array}{ccccc}
\mathsf{T}(!\mathsf{T}A \otimes \mathsf{T}A) & \xrightarrow{\mathsf{n}_2} & \mathsf{T}!\mathsf{T}A \otimes \mathsf{TT}A & \xrightarrow{\mu^\sharp \otimes \mu} & !\mathsf{T}A \otimes \mathsf{T}A \\
\mathsf{T}(\mathsf{d})\downarrow & & & & \downarrow \mathsf{d} \\
\mathsf{T}!\mathsf{T}A & & \xrightarrow{\hspace{8em}\mu^\sharp\hspace{8em}} & & !\mathsf{T}A
\end{array} \quad (28)
$$

Consider now the induced mixed distributive law λ of $(\mathsf{T}, \mu, \eta, \mathsf{n}_2, \mathsf{n}_1)$ over $(!, \delta, \varepsilon, \Delta, \mathsf{e})$ from Proposition 11. Then that λ satisfies the extra necessary condition follows from commutativity of the following diagram:

$$
\begin{array}{ccccccc}
\mathsf{T}(!A \otimes A) & \xrightarrow{\mathsf{n}_2} & \mathsf{T}!A \otimes \mathsf{T}A & \xrightarrow{\mathsf{T}!(\eta) \otimes 1} & \mathsf{T}!\mathsf{T}A \otimes \mathsf{T}A & \xrightarrow{\mu^\sharp \otimes 1} & !\mathsf{T}A \otimes \mathsf{T}A \\
\mathsf{d}\downarrow & \searrow^{\mathsf{T}(!(\eta) \otimes \eta)} & \text{Nat. of. } \mathsf{n}_2 & & 1 \otimes \mathsf{T}(\eta)\downarrow & (1)\;\nearrow^{\mu^\sharp \otimes \mu} & \downarrow \mathsf{d} \\
& & \mathsf{T}(!\mathsf{T}A \otimes \mathsf{T}A) & \xrightarrow{\mathsf{n}_2} & \mathsf{T}!\mathsf{T}A \otimes \mathsf{TT}A & & \\
& \text{Nat. of. } \mathsf{d} & \mathsf{T}(\mathsf{d})\downarrow & & (28) & & \\
\mathsf{T}(!A) & \xrightarrow{\mathsf{T}!(\eta)} & \mathsf{T}!\mathsf{T}A & \xrightarrow{\hspace{6em}\mu^\sharp\hspace{6em}} & & & !\mathsf{T}A
\end{array}
$$

◀

Internal Universes in Models of Homotopy Type Theory

Daniel R. Licata[1]

Wesleyan University Dept. Mathematics & Computer Science, Middletown, USA

 https://orcid.org/0000-0003-0697-7405

Ian Orton[2]

University of Cambridge Dept. Computer Science & Technology, Cambridge, UK

 https://orcid.org/0000-0002-9924-0623

Andrew M. Pitts

University of Cambridge Dept. Computer Science & Technology, Cambridge, UK

 https://orcid.org/0000-0001-7775-3471

Bas Spitters[3]

Aarhus University Dept. Computer Science, Aarhus, DK

 https://orcid.org/0000-0002-2802-0973

───── **Abstract** ─────

We begin by recalling the essentially global character of universes in various models of homotopy type theory, which prevents a straightforward axiomatization of their properties using the internal language of the presheaf toposes from which these model are constructed. We get around this problem by extending the internal language with a modal operator for expressing properties of global elements. In this setting we show how to construct a universe that classifies the Cohen-Coquand-Huber-Mörtberg (CCHM) notion of fibration from their cubical sets model, starting from the assumption that the interval is tiny—a property that the interval in cubical sets does indeed have. This leads to an elementary axiomatization of that and related models of homotopy type theory within what we call *crisp type theory*.

---

1   Introduction

Voevodsky's univalence axiom in Homotopy Type Theory (HoTT) [39] is motivated by the fact that constructions on structured types should be invariant under isomorphism. From a programming point of view, such constructions can be seen as type-generic programs. For example, if G and H are isomorphic groups, then for any construction C on groups, an instance $C(G)$ can be *transported* to $C(H)$ by lifting this isomorphism using a type-generic program corresponding to C. As things stand, there is no single definition of the semantics of such generic programs; instead there are several variations on the theme of giving a computational interpretation to the new primitives of HoTT (univalence and higher inductive types) via different constructive models [9, 13, 6, 5], the pros and cons of which are still being explored.

As we show in this paper, that exploration benefits from being carried out in a type-theoretic language. This is different from developing the consequences of HoTT itself using a type-theoretic language, such as intensional Martin-Löf type theory with axioms for univalence and higher inductive types, as used in [39]. There *all* types have higher-dimensional structure, or "are fibrant" as one says, via the structure of the iterated identity types associated with them. Contrastingly, when using type theory to describe models of HoTT, being fibrant is an explicit structure external to a type; and that structure can itself be classified by a type, so that users of the type theory can *prove* that a type is fibrant by inhabiting a certain other type. As an example, consider the cubical sets model of type theory introduced by Cohen, Coquand, Huber and Mörtberg (CCHM) [13]. This model uses a presheaf topos on a particular category of cubes that we denote by $\square$, generated by an interval object $\mathbb{I}$, maps out of which represent paths. The corresponding presheaf topos $\hat{\square}$ has an associated category with families (CwF) [15] structure that gives a model of Extensional Martin-Löf Type Theory [27] in a standard way [19]. While not all types in this presheaf topos have a fibration structure in the CCHM sense, working within constructive set theory, CCHM show how to make a new CwF of fibrant types out of this presheaf CwF, one which is a model of Intensional Martin-Löf Type Theory with univalent universes and (some) higher inductive types [39]. Their model construction is rather subtle and complicated. Coquand noticed that the CCHM version of Kan fibration could be more simply described in terms of partial elements in the *internal language* of the topos. Some of us took up and expanded upon that suggestion in [30] and [10, Section 4]. Using Extensional Martin-Löf Type Theory with an impredicative universe of propositions (one candidate for the internal language of toposes), those works identify some relatively simple axioms for an interval and a collection of Kan-filling shapes (*cofibrant* propositions) that are sufficient to define a CwF of CCHM fibrations and prove most of its properties as a model of univalent foundations, for example, that Π, Σ, path and other types are fibrant. These internal language constructions can be used as an intermediate point in constructing a concrete model in cubical sets: the type theory of HoTT [39] can be translated into the internal language of the topos, which has a semantics in the topos itself in a standard way. The advantages of this indirection are two-fold. First, the definition and properties of the notion of fibration (both the CCHM notion [13] and other related ones [5, 34]) are simpler when expressed in the internal language; and secondly, so long as the axioms are not too constraining, it opens up the possibility of finding new models of HoTT. Indeed, since our axioms do not rely on the infinitary aspects of Grothendieck toposes (such as having infinite colimits), it is possible to consider models of them in elementary toposes, such as Hyland's effective topos [16, 38].

From another point of view, the internal language of the presheaf topos can itself be viewed as a two-level type theory [4, 40] with fibrant and non-fibrant types, where being fibrant is classified by a type, and the constructions are a library of fibrancy instances for all of the usual types of type theory. Directed type theory [34] has a very similar story: it adds a directed interval type and a logic of partial elements to homotopy type theory, and using them defines some new notions of higher-dimensional structure, including co- and contravariant fibrations.

However, the existing work describing models using an internal language [30, 10, 5] does not encompass *universes* of fibrant types. The lack of universes is a glaring omission for making models of HoTT, due to both their importance and the difficulty of defining them correctly. Moreover, it is an impediment to using internal language presentations of cubical type theory as a two-level type theory. For example, most constructions on higher inductive types, like calculating their homotopy groups, require a fibrant universe of fibrant types; and adding universes to directed type theory would have analogous applications. Finally, packaging the fibrant types together into a universe restores much of the convenience of working in a language where all types are fibrant: instead of passing around separate fibrancy proofs, one knows that a type is fibrant by virtue of the universe to which it belongs.

In this paper, we address this issue by studying universes of fibrant types expressed in internal languages for models of cubical type theories. CCHM [13] define a universe concretely using a version of the Hofmann-Streicher universe construction in presheaf toposes [20]. This gives a classifier for their notion of fibration—the universe is equipped with a CCHM fibration that gives rise to every fibration (with small fibres) by re-indexing along a function into the universe. In this way one gets a model of a Tarski-style universe closed under whatever type-forming operations are supported by CCHM fibrations. Thus, there is an appropriate semantic target for a universe of fibrant types, but neither [30], nor [10] gave a version of such a universe expressed in the internal language. This is for a good reason: [32, Remark 7.5] points out that there can be no *internal* universe of types equipped with a CCHM fibration that weakly classifies fibrations. We recall in detail why this is the case in Section 3, but the essence is that naïve axioms for a weak classifier for fibrations imply that a family of types, each member of which is fibrant, has to form a fibrant family; but this is not true for many notions of fibration, such as the CCHM one.

To fix this issue, in Section 4 we enrich the internal language to a *modal* type theory with two context zones [33, 14, 36], inspired in particular by the fact that cubical sets are a model of Shulman's spatial type theory. In a judgement $\Delta \mid \Gamma \vdash a : A$ of this modal type theory, the context Γ represents the usual local elements of types in the topos, while the new context Δ represents global ones. The dual context structure is that of an S4 necessity modality in modal logic, because a global element determines a local one, but global elements cannot refer to local elements. We use Shulman's term "crisp" for variables from Δ, and call the type theory *crisp type theory*, because we do not in fact use any of the modal type operators of his spatial type theory, but just Π-types whose domains are crisp. Using these crisp Π-types, we give axioms that specify a universe that classifies global fibrations—the modal structure forbids the internal substitutions that led to inconsistency.

One approach to validating these universe axioms would be to check them directly in a cubical set model; but we can in fact do more work using crisp type theory as the internal language and reduce the universe axioms to a structure that is simpler to check in models. Specifically, in Theorem 5.2, we construct such a universe from the assumption that the interval $\mathbb{I}$ is *tiny*, which by definition means that its exponential functor $\mathbb{I} \to \_$ has a right adjoint (a global one, not an internal one—this is another example where crisp type theory is needed to express this distinction). The ubiquity of right adjoints to exponential functors was

first pointed out by Lawvere [23] in the context of synthetic differential geometry. Awodey pointed out their occurrence in interval-based models of type theory in his work on various cube categories [7]. As far as we know, it was Sattler who first suggested their relevance to constructing universes in such models (see [35, Remark 8.3]). It is indeed the case that the interval object in the topos of cubical sets is tiny. Some ingenuity is needed to use the right adjoint to $\mathbb{I} \to \_$ to construct a universe with a fibration that gives rise to every other one up to equality, rather than just up to isomorphism; we employ a technique of Voevodsky [41] to do so.

Finally, we describe briefly some applications in Section 6. First, our universe construction based on a tiny interval is the missing piece that allows a completely internal development of a model of univalent foundations based upon the CCHM notion of fibration, albeit internal to crisp type theory rather than ordinary type theory. Secondly, we describe a preliminary result showing that our axioms for universes are suitable for building type theories with hierarchies of universes, each with a different notion of fibration.

The constructions and proofs in this paper have been formalized in Agda-flat [2], an appealingly simple extension of Agda [3] that implements crisp type theory; see `https://doi.org/10.17863/CAM.22369`. Agda-flat was provided to us by Vezzosi as a by-product of his work on modal type theory and parametricity [29].

2   Internal description of fibrations

We begin by recalling from [32, 10] the internal description of fibrations in presheaf models, using CCHM fibrations [13, Definition 13] as an example. Rather than using Extensional Martin-Löf Type Theory with an impredicative universe of propositions as in [32, 10], here we use an intensional and predicative version, therefore keeping within a type theory with decidable judgements.[4] Our type theory of choice is the one implemented by Agda [3], whose assistance we have found invaluable for developing and checking the definitions. Adopting Agda-style syntax, dependent function types are written $(x : A) \to B\,x$, or $\{x : A\} \to B\,x$ if the argument to the function is implicit; non-dependent function types are written $(\_ : A) \to B$, or just $A \to B$. There is a non-cumulative hierarchy of Russell-style [25] universe types $\mathsf{Set} = \mathsf{Set}_0 : \mathsf{Set}_1 : \mathsf{Set}_2 : \mathsf{Set}_3 \ldots$ Among Agda's inductive types we need identity types $\_ \equiv \_ : \{A : \mathsf{Set}_n\} \to A \to A \to \mathsf{Set}_n$, which form the inductively defined family of types with a single constructor $\mathsf{refl} : \{A : \mathsf{Set}_n\}\{x : A\} \to x \equiv x$; and we need the empty inductive type $\bot : \mathsf{Set}$, which has no constructors. Among Agda's record types (inductive types with a single constructor for which η-expansion holds definitionally) we need the unit type $\top : \mathsf{Set}$ with constructor $\mathsf{tt} : \top$; and dependent products (Σ-types), that we write as $\Sigma\,x : A\,,\,B\,x$ and which are dependent record types with constructor $(\_\,,\_) : (x : A)(\_ : B\,x) \to \Sigma\,x : A\,,\,B\,x$ and fields (projections) $\mathsf{fst} : (\Sigma\,x : A\,,\,B\,x) \to A$ and $\mathsf{snd} : (z : \Sigma\,x : A\,,\,B\,x) \to B(\mathsf{fst}\,z)$.

This type theory can be interpreted in (the category with families of) any presheaf topos, such as the one defined below, so long as we assume that the ambient set theory has a countable hierarchy of Grothendieck universes; in particular, one could use a constructive ambient set theory such as IZF [1] with universes. We will use the fact that the interpretation of the type theory in presheaf toposes satisfies *function extensionality* and *uniqueness of identity proofs*:

$$\mathsf{funext} : \{A : \mathsf{Set}_n\}\{B : A \to \mathsf{Set}_m\}\{f\ g : (x : A) \to B\,x\}((x : A) \to f\,x \equiv g\,x) \to f \equiv g \quad (1)$$

$$\mathsf{uip} : \{A : \mathsf{Set}_n\}\{x\ y : A\}\{p\ q : x \equiv y\} \to p \equiv q \qquad\qquad\qquad\qquad\qquad (2)$$

---

[4] Albeit at the expense of some calculations with universe levels; Coq's universe polymorphism would probably deal with this aspect automatically.

▶ **Definition 2.1** (Presheaf topos of de Morgan cubical sets)**.** Let $\square$ denote the small category with finite products which is the Lawvere theory of De Morgan algebra (see [8, Chap. XI] and [37, Section 2]). Concretely, $\square^{\mathrm{op}}$ consists of the free De Morgan algebras on n generators, for each $n \in \mathbb{N}$, and the homomorphisms between them. Thus $\square$ contains an object I that generates the others by taking finite products, namely the free De Morgan algebra on one generator. This object is the generic De Morgan algebra and in particular it has two distinct global elements, corresponding to the constants for the greatest and least elements. The *topos of cubical sets* [13], which we denote by $\widehat{\square}$, is the category of $\mathcal{S}et$-valued functors on $\square^{\mathrm{op}}$ and natural transformations between them. The Yoneda embedding, written $\mathrm{y} : \square \hookrightarrow \widehat{\square}$, sends $\mathrm{I} \in \square$ with its two distinct global elements to a representable presheaf $\mathbb{I} = \mathrm{yI}$ with two distinct global elements. This interval $\mathbb{I}$ is used to model path types: a path in A from a_0 to a_1 is any morphism $\mathbb{I} \to A$ that when composed with the distinct global elements gives a_0 and a_1.

The toposes used in other cubical models [9, 6, 5] vary the choice of algebra from the De Morgan case used above; see [11]. To describe all these cubical models using type theory as an internal language, we postulate the existence of an *interval* type $\mathbb{I}$ with two distinct elements, which we write as $\mathtt{0}$ and $\mathtt{I}$:

$$\mathbb{I} : \mathsf{Set} \qquad \mathtt{0} : \mathbb{I} \qquad \mathtt{I} : \mathbb{I} \qquad \mathtt{0} \not\equiv \mathtt{I} : (\mathtt{0} \equiv \mathtt{I}) \to \bot \tag{3}$$

Apart from an interval, the other data needed to define a cubical sets model of homotopy type theory is a notion of *cofibration*, which specifies the shapes of filling problems that can be solved in a dependent type. For this, CCHM [13] use a particular subobject of $\Omega \in \widehat{\square}$ (the subobject classifier in the topos $\widehat{\square}$), called the *face lattice*; but other choices are possible [32]. Here, we avoid the use of the impredicative universe of propositions Ω and just assume the existence of a collection of "cofibrant" types in the first universe Set, including at least the empty type $\bot$ (in Section 6, we will introduce more cofibrations, needed to model various type constructs):

$$\mathsf{cof} : \mathsf{Set} \to \mathsf{Set} \qquad \mathsf{cof}_\bot : \mathsf{cof}\,\bot \tag{4}$$

We call $\varphi : \mathsf{Set}$ *cofibrant* if $\mathsf{cof}\,\varphi$ holds, that is, if we can supply a term of that type. To define the fibrations as a type in the internal language we use two pieces of notation. First, the *path functor* associated with the interval $\mathbb{I}$ is

$$\begin{aligned} &\wp : \mathsf{Set}_n \to \mathsf{Set}_n & &\wp\,{}^{\backprime} : \{A : \mathsf{Set}_n\}\{B : \mathsf{Set}_m\}(f : A \to B) \to \wp\,A \to \wp\,B \\ &\wp\,A = \mathbb{I} \to A & &\wp\,{}^{\backprime}f\,p\,i = f(p\,i) \end{aligned} \tag{5}$$

Secondly, we define the following *extension* relation

$$\_\,\nearrow\,\_ : \{\varphi : \mathsf{Set}\}\{A : \mathsf{Set}_n\}(t : \varphi \to A)(x : A) \to \mathsf{Set}_n \qquad t \nearrow x = (u : \varphi) \to t\,u \equiv x \tag{6}$$

Thus $t \nearrow x$ is the type of proofs that the partial element $t : \varphi \to A$ extends to the (total) element $x : A$. We will use this when t denotes a partial element of A of cofibrant extent, that is when we have a proof of $\mathsf{cof}\,\varphi$.

▶ **Definition 2.2** (fibrations)**.** The type $\mathsf{isFib}\,A$ of *fibration structures* for a family of types $A : \Gamma \to \mathsf{Set}_n$ over some type $\Gamma : \mathsf{Set}_m$ consists of functions taking any path $p : \wp\,\Gamma$ in the base type to a *composition structure* in $\mathsf{C}(A \circ p)$:

$$\mathsf{isFib} : (\Gamma : \mathsf{Set}_m)(A : \Gamma \to \mathsf{Set}_n) \to \mathsf{Set}_{1 \sqcup m \sqcup n} \qquad \mathsf{isFib}\,\Gamma\,A = (p : \wp\,\Gamma) \to \mathsf{C}(A \circ p) \tag{7}$$

Here C is some given function $\wp\,\mathsf{Set}_n \to \mathsf{Set}_{1\sqcup n}$ (polymorphic in the universe level n) which parameterizes the notion of fibration. Then for each type Γ, the type $\mathsf{Fib}_n\,\Gamma$ of *fibrations* over it with fibers in Set_n consists of families equipped with a fibration structure

$$\mathsf{Fib}_n : (\Gamma : \mathsf{Set}_m) \to \mathsf{Set}_{m\sqcup(n+1)} \qquad \mathsf{Fib}_n\,\Gamma = \Sigma\,A : (\Gamma \to \mathsf{Set}_n)\,,\, \mathsf{isFib}\,\Gamma\,A \tag{8}$$

and there are *re-indexing* functions, given by composition of dependent functions $(\_ \circ \_)$

$$\_[\_] : \{\Gamma : \mathsf{Set}_k\}\{\Gamma' : \mathsf{Set}_m\}(\Phi : \mathsf{Fib}_n\,\Gamma)(\gamma : \Gamma' \to \Gamma) \to \mathsf{Fib}_n\,\Gamma' \tag{9}$$
$$(A\,,\,\alpha)[\gamma] = (A \circ f\,,\,\alpha \circ \wp^{\iota}\,f)$$

A *CCHM fibration* is the above notion of fibration for the composition structure $\mathsf{CCHM} : \wp\,\mathsf{Set}_n \to \mathsf{Set}_{1\sqcup n}$ from [13]:

$$\mathsf{CCHM}\,P = (\varphi : \mathsf{Set})(\_ : \mathsf{cof}\,\varphi)(p : (i : \mathbb{I}) \to \varphi \to P\,i) \to (\Sigma\,a_0 : P\,\mathtt{0}\,,\,p\,\mathtt{0} \nearrow a_0) \to \tag{10}$$
$$(\Sigma\,a_1 : P\,\mathtt{1}\,,\,p\,\mathtt{1} \nearrow a_1)$$

Thus the type $\mathsf{CCHM}\,P$ of CCHM composition structures for a path of types $P : \wp\,\mathsf{Set}_n$ consists of functions taking any dependently-typed path of partial elements $p : (i : \mathbb{I}) \to \varphi \to P\,i$ of cofibrant extent to a function mapping extensions of the path at one end $p\,\mathtt{0} \nearrow a_0$, to extensions of it at the other end $p\,\mathtt{1} \nearrow a_1$. When the cofibration is $\bot$, this $\mathsf{isFib}\,\Gamma\,A$ expands to the statement that for all paths $p : \mathbb{I} \to \Gamma$, $A(p\,\mathtt{0}) \to A(p\,\mathtt{1})$, so that this internal language type says that A is equipped with a transport function along paths in Γ. The use of cofibrant partial elements generalizes transport with a notion of path composition, which is used to show that path types are fibrant.

Other notions of fibration follow the above definitions but vary the definition of $\mathsf{C} : \wp\,\mathsf{Set}_n \to \mathsf{Set}_{1\sqcup n}$; for example, generalized diagonal Kan composition [5]. Co/contravariant fibrations in directed type theory [34] also have the form of isFib for some C, but with $\wp$ being directed paths. Definition 2.2 illustrates the advantages of internal-language presentations; in particular, uniformity [13] is automatic.

If Γ denotes an object of the cubical sets topos $\widehat{\square}$, then $\mathsf{Fib}_0\,\Gamma$ denotes an object whose global sections correspond to the elements of the set $\mathrm{FTy}(\Gamma)$ of families over Γ equipped with a composition structure as defined in [13, Definition 13]. Our goal now is to first recall that there can be no universe that weakly classifies these CCHM fibrations in an internal sense, and then move to a modal type theory where such a universe can be expressed.

3   The "no-go" theorem for internal universes

In this section we recall from [32, Remark 7.5] why there can be no universe that weakly classifies CCHM fibrations in an internal sense. Such a weak classifier would be given by the following data

$$\mathsf{U} : \mathsf{Set}_2 \qquad\qquad \mathsf{code} : \{\Gamma : \mathsf{Set}\}(\Phi : \mathsf{Fib}_0\,\Gamma) \to \Gamma \to \mathsf{U}$$
$$\mathsf{El} : \mathsf{Fib}_0\,\mathsf{U} \qquad\qquad \mathsf{Elcode} : \{\Gamma : \mathsf{Set}\}(\Phi : \mathsf{Fib}_0\,\Gamma) \to \mathsf{El}[\mathsf{code}\,\Phi] \equiv \Phi \tag{11}$$

where for simplicity we restrict attention to fibrations whose fibers are in the lowest universe, $\mathsf{Set} = \mathsf{Set}_0$. Here U is the universe[5] and El is a CCHM fibration over it which is a weak

---

[5] Our predicative treatment of cofibrant types makes it necessary to place U in Set_2 rather than Set_1.

classifier in the sense that any fibration $\Phi : \mathsf{Fib}_0\,\Gamma$ can be obtained from it (up to equality) by re-indexing along some function $\mathsf{code}\,\Phi : \Gamma \to \mathsf{U}$. (The word "weak" refers to the fact that we do not require there to be a *unique* function $\gamma : \Gamma \to \mathsf{U}$ with $\mathsf{El}[\gamma] \equiv \Phi$.)

We will show that the data in (11) implies that the interval must be trivial ($\mathsf{O} \equiv \mathsf{I}$), contradicting the assumption in (3). This is because (11) allows one to deduce that if a family of types $A : \Gamma \to \mathsf{Set}$ has the property that each $A\,x$ has a fibration structure when regarded as a family over the unit type $\top$, then there is a fibration structure for the whole family A; and yet there are families where this cannot be the case. For example, consider the family $P : \mathbb{I} \to \mathsf{Set}$ with $P\,i = (\mathsf{O} \equiv i)$. For each $i : \mathbb{I}$, the type $P\,i$ has a fibration structure $\pi\,i : \mathsf{isFib}\,\top\,(\lambda\,\_ \to P\,i)$, because of uniqueness of identity proofs (2). But the family as a whole satisfies $\mathsf{isFib}\,\mathbb{I}\,P \to \bot$, because if we had a fibration structure $\alpha : \mathsf{isFib}\,\mathbb{I}\,P$, then we could apply it to

$$\mathsf{id} : \wp\,\mathbb{I} \qquad \varphi : \mathsf{Set} \qquad u : \mathsf{cof}\,\varphi \qquad p : (i : \mathbb{I}) \to \varphi \to P\,i \qquad z : \Sigma\,a_0 : P\,\mathsf{O}\,,\,p\,\mathsf{O} \nearrow a_0$$
$$\mathsf{id}\,i = i \qquad \varphi = \bot \qquad u = \mathsf{cof}_\bot \qquad p\,i = \lambda\,\_ \to \bot\mathsf{elim} \qquad z = (\mathsf{refl}\,,\,\lambda\,\_ \to \mathsf{uip})$$

(where $\bot\mathsf{elim} : \{A : \mathsf{Set}\} \to \bot \to A$ is the elimination function for the empty type) to get $\alpha\,\mathsf{id}\,\varphi\,u\,p\,z : (\Sigma\,a_1 : P\,\mathsf{I}\,,\,p\,\mathsf{I} \nearrow a_1)$ and hence $\mathsf{O} \not\equiv \mathsf{I}\,(\mathsf{fst}\,(\alpha\,\mathsf{id}\,\varphi\,u\,p\,z)) : \bot$. From this we deduce the following "no-go" [6] theorem for internal universes of CCHM fibrations.

▶ **Theorem 3.1.** [32, Remark 7.5] *The existence of types and functions as in* (11) *for CCHM fibrations is contradictory. More precisely, if* $\mathsf{IntUniv} : \mathsf{Set}_3$ *is the dependent record type with fields* U, El, code *and* Elcode *as in* (11)*, then there is a term of type* $\mathsf{IntUniv} \to \bot$*.*

Proof. [7] Suppose we have an element of $\mathsf{IntUniv}$ and hence functions as in (11). Then taking P to be $\lambda\,i \to (\mathsf{O} \equiv i)$ and using the family $\pi\,i$ of fibration structures on each type $P\,i$ mentioned above, we get:

$$\Phi : \mathsf{Fib}_0\,\mathbb{I} \qquad \Phi = \mathsf{El}[(\lambda\,i \to \mathsf{code}\,((\lambda\,\_ \to P\,i)\,,\,\pi\,i)\,\mathsf{tt})] \tag{12}$$

Using Elcode and function extensionality (1), it follows that there is a proof $u : \mathsf{fst}\,\Phi \equiv P$, namely $u = \mathsf{funext}\,(\lambda\,i \to \mathsf{cong}\,(\lambda\,x \to \mathsf{fst}\,x\,\mathsf{tt})\,(\mathsf{Elcode}\,((\lambda\,\_ \to P\,i)\,,\,\pi\,i)))$, where cong is the usual congruence property of equality. From that and $\mathsf{snd}\,\Phi$ we get an element of $\mathsf{isFib}\,\mathbb{I}\,P$. But we saw above how to transform such an element into a proof of $\bot$. So altogether we have a proof of $\mathsf{IntUniv} \to \bot$. ◀

▶ **Remark 3.2.** This counterexample generalizes to other notions of fibration: it is not usually the case that any type family $A : \Gamma \to \mathsf{Set}$ for which $A\,x$ is fibrant over $\top$ for all $x : \Gamma$, is fibrant over Γ. The above proof should be compared with the proof that there is no "fibrant replacement" type-former in *Homotopy Type System* (HTS); see `https://ncatlab.org/homotopytypetheory/show/Homotopy+Type+System#fibrant_replacement`. Theorem 5.1 below provides a further example of a global construct that does not internalize.

4 Crisp type theory

The proof of Theorem 3.1 depends upon the fact that in the internal language the `code` function can be applied to elements with free variables. In this case it is the variable $i : \mathbb{I}$ in $\mathsf{code}\,((\lambda\,\_ \to P\,i)\,,\,\pi\,i)\,\mathsf{tt}$; by abstracting over it we get a function $\mathbb{I} \to \mathsf{U}$ and re-indexing El

---

[6] We are stealing Shulman's terminology [36, section 4.1].

[7] See the file `theorem-3-1.agda` at `https://doi.org/10.17863/CAM.22369` for an Agda version of this proof.

along this function gives the offending fibration (12). Nevertheless, the cubical sets presheaf topos does contain a (univalent) universe which is a CCHM fibration classifier, but only in an *external* sense. Thus there is an object U in $\widehat{\square}$ and a global section $\mathsf{El} : 1 \to \mathsf{Fib}_0\,\mathsf{U}$ with the property that for any object Γ and morphism $\Phi : 1 \to \mathsf{Fib}_0\,\Gamma$, there is a morphism $\mathsf{code}\,\Phi : \Gamma \to \mathsf{U}$ so that Φ is equal to the composition $\mathsf{Fib}_0\,(\mathsf{code}\,\Phi) \circ \mathsf{El} : 1 \to \mathsf{Fib}_0\,\Gamma$; see [13, Definition 18] for a concrete description of U. The internalization of this property replaces the use of global elements $1 \to \Gamma$ of an object by local elements, that is, morphisms $X \to \Gamma$ where X ranges over a suitable collection of generating objects (for example, the representable objects in a presheaf topos); and we have seen that such an internalized version cannot exist.

Nevertheless, we would like to explain the construction of universes like $\mathsf{U} \in \widehat{\square}$ using some kind of type-theoretic language that builds on Section 2. So we seek a way of manipulating global elements of an object Γ, within the internal language. One cannot do so simply by quantifying over elements of the type $\top \to \Gamma$, because of the isomorphism $\Gamma \cong (\top \to \Gamma)$. Instead, we pass to a modal type theory that can speak about global elements, which we call *crisp type theory*. Its judgements, such as $\Delta \mid \Gamma \vdash a : A$, have two context zones—where Δ represents global elements and Γ the usual, local ones. The context structure is that used for an S4 necessitation modality [33, 14, 36], because a global element from Δ can be used locally, but global elements cannot depend on local variables from Γ. Following [36], we say that the left-hand context Δ contains *crisp* hypotheses about the types of variables, written $x :: A$.

The interpretation of crisp type theory in cubical sets makes use of the comonad $\flat : \widehat{\square} \to \widehat{\square}$ that sends a presheaf A to the constant presheaf on the set of global sections of A; thus $\flat A(X) \cong A(1)$ for all $X \in \square$ (where $1 \in \square$ is terminal). Then a judgement $\Delta \mid \Gamma \vdash a : A$ describes the situation where Δ is a presheaf, Γ is a family of presheaves over $\flat\Delta$, A is a family over $\Sigma(\flat\Delta)\,\Gamma$ and a is an element of that family. The rules of crisp type theory are designed to be sound for this interpretation. Compared with ordinary type theory, the key constraint is that *types in the crisp context and terms substituted for crisp variables depend only on crisp variables.* The crisp variable and (admissible) substitution rules are:

$$\frac{}{\Delta, x :: A, \Delta' \mid \Gamma \vdash x : A} \qquad \frac{\Delta \mid \diamond \vdash a : A \qquad \Delta, x :: A, \Delta' \mid \Gamma \vdash b : B}{\Delta, \Delta'[a/x] \mid \Gamma[a/x] \vdash b[a/x] : B[a/x]} \tag{13}$$

The semantics of the variable rule, which says that global elements can be used locally, uses the counit $\varepsilon\,A : \flat A \to A$ of the comonad $\flat$ mentioned above. In the substitution rule, $\diamond$ stands for the empty list, so a and A may only depend upon the crisp variables from Δ. The other rules of crisp type theory (those for Π types, Σ types, etc.) carry the crisp context along. For our application we do not need a type-former for $\flat$, but instead make use of crisp Π types (see, e.g. [14, 28]), that is, Π types whose domain is crisp

$$\frac{\begin{array}{c}\Delta \mid \diamond \vdash A : \mathsf{Set}_m \\ \Delta, x :: A \mid \Gamma \vdash B : \mathsf{Set}_n\end{array}}{\Delta \mid \Gamma \vdash (x :: A) \to B : \mathsf{Set}_{m \sqcup n}} \qquad \frac{\Delta, x :: A \mid \Gamma \vdash b : B}{\Delta \mid \Gamma \vdash \lambda x :: A.b : (x :: A) \to B} \qquad \frac{\begin{array}{c}\Delta \mid \Gamma \vdash f : (x :: A) \to B \\ \Delta \mid \diamond \vdash a : A\end{array}}{\Delta \mid \Gamma \vdash f\,a : B[a/x]} \tag{14}$$

with $\beta\eta$ judgemental equalities. In these rules, because the argument variable x is crisp, its type A, and the term a to which the function f is applied, must also be crisp. We also use *crisp induction* for identity types [36]—identity elimination with a family $y :: A, p :: x \equiv y \vdash C(y, p)$ whose parameters are crisp variables, which is given by a term of type

$$\{A :: \mathsf{Set}_n\}\{x :: A\}(C : (y :: A)(p :: x \equiv y) \to \mathsf{Set}_n)(z : C\,x\,\mathsf{refl})(y :: A)(p :: x \equiv y) \to C\,y\,p \tag{15}$$

together with a β judgemental equality.

▶ Remark 4.1 (Presheaf models of crisp type theory). Crisp type theory is motivated by the specific presheaf topos $\widehat{\square}$ from Definition 2.1. However, it seems that very little is required of a category **C** for the presheaf topos $\widehat{\mathbf{C}}$ to soundly interpret it using the comonad $\flat = p^* \circ p_*$, where p_* takes the global sections of a presheaf and its left adjoint p^* sends sets to constant presheaves. This $\flat$ preserves finite limits (because it is the composition of functors with left adjoints—p^* is isomorphic to the functor given by precomposition with $\mathbf{C} \to 1$ and hence has a left adjoint given by left Kan extension along $\mathbf{C} \to 1$). Although the details remain to be worked out, it appears that to model crisp type theory with crisp Π types and crisp identification induction (and moreover a $\flat$ modality with crisp $\flat$ induction, which we do not use here), the only additional condition needed is that this comonad is idempotent (meaning that the comultiplication $\delta : \flat \to \flat \circ \flat$ is an isomorphism). This idempotence holds iff $\widehat{\mathbf{C}}$ is a connected topos, which is the case iff **C** is a connected category—for example, when **C** has a terminal object. If it does have a terminal object, then $\widehat{\mathbf{C}}$ is a local topos [21, Sect. C3.6] and $\flat$ has a right adjoint; in which case, conjecturally [36, Remark 7.5], one gets a model of the whole of Shulman's spatial type theory, of which crisp type theory is a part. In fact $\widehat{\square}$ does not just have a terminal object, it has all finite products (as does any Lawvere theory) and from this it follows that $\widehat{\square}$ is not just local, but also cohesive [24].

▶ Remark 4.2 (Agda-flat). Vezzosi has created a fork of Agda, called *Agda-flat* [2], which allows us to explore crisp type theory. It adds the ability to use crisp variables[8] $x :: A$ in places where ordinary variables $x : A$ may occur in Agda, and checks the modal restrictions in the above rules. For example, Agda-flat quite correctly rejects the following attempted application of a crisp-Π function to an ordinary argument

$$\mathsf{wrong} : (A :: \mathsf{Set}_n)(B : \mathsf{Set}_m)(f : (\_ :: A) \to B)(x : A) \to B \qquad \mathsf{wrong}\ A\ B\ f\ x = f(x)$$

while the variant with $x :: A$ succeeds. This is a simple example of keeping to the modal discipline that crisp type theory imposes; for more complicated cases, such as occur in the proof of Theorem 5.2 below, we have found Agda-flat indispensable for avoiding errors. However, Agda-flat implements a superset of crisp type theory and more work is needed to understand their precise relationship. For example, Agda's ability to define inductive types leads to new types in Agda-flat, such as the $\flat$ modality itself; and its pattern-matching facilities allow one to prove properties of $\flat$ that go beyond crisp type theory. Agda allows one to switch off pattern-matching in a module; to be safe we do that as far as possible in our development. Installation instructions for Agda-flat can be found at `https://doi.org/10.17863/CAM.22369`.

5   Universes from tiny intervals

In crisp type theory, to avoid the inconsistency in the "no-go" Theorem 3.1, we can weaken the definition of a universe in (11) by taking code and Elcode to be crisp functions of fibrations Φ (and implicitly, of the base type Γ of the fibration). For if code has type $\{\Gamma :: \mathsf{Set}\}(\Phi :: \mathsf{Fib}_0\ \Gamma)(x : \Gamma) \to \mathsf{U}$, then the proof of a contradiction is blocked when in (12) we try to apply code to $\Phi = ((\lambda\, \_ \to P\, i)\, , \pi\, i)$, which depends upon the local variable $i : \mathbb{I}$. Indeed we show in this section that given an extra assumption about the interval type $\mathbb{I}$ that holds for cubical sets, it is possible to define a universe with such crisp coding functions which moreover are unique, so that one gets a classifying fibration, rather than just a weakly classifying one.

---

[8] The Agda-flat concrete syntax for "$x :: A$" is "$x :\{\flat\}\ A$".

Recall from Definition 2.1 that in the cubical sets model, the type $\mathbb{I}$ denotes the representable presheaf $y\mathrm{I} \in \widehat{\square}$ on the object $\mathrm{I} \in \square$. Since $\square$ has finite products, there is a functor $\_ \times \mathrm{I} : \square \to \square$. Pre-composition with this functor induces an endofunctor on presheaves $(\_ \times \mathrm{I})^* : \widehat{\square} \to \widehat{\square}$ which has left and right adjoints, given by left and right Kan extension [26, Chap. X] along $\_ \times \mathrm{I}$. Hence by the Yoneda Lemma, for any $F \in \widehat{\square}$ and $X \in \square$

$$(\mathbb{I} \to F)\, X \cong \widehat{\square}(yX, \mathbb{I} \to F) \cong \widehat{\square}(yX \times y\mathrm{I}, F) \cong \widehat{\square}(y(X \times \mathrm{I}), F) = ((\_ \times \mathrm{I})^* F)\, X$$

naturally in both X and F. It follows that the exponential functor $\wp = \mathbb{I} \to \_ : \widehat{\square} \to \widehat{\square}$ is naturally isomorphic to $(\_ \times \mathrm{I})^*$ and hence not only has a left adjoint (corresponding to product with $\mathbb{I}$) but also a right adjoint. The significance of objects in a category with finite products that are not only exponentiable (product with them has a right adjoint), but also whose exponential functor has a right adjoint was first pointed out by Lawvere in the context of synthetic differential geometry [23]. He called such objects "atomic", but we will follow later usage [42] and call them *tiny*.[9] Thus the interval in cubical sets is tiny and we have a right adjoint to the path functor $\wp$ that we denote by $\sqrt{\ } : \widehat{\square} \to \widehat{\square}$. So for each $B \in \widehat{\square}$, the functor $\widehat{\square}(\wp\,\_, B) : \widehat{\square} \to \mathcal{S}et$ is representable by $\sqrt{B}$, that is, there are bijections $\widehat{\square}(\wp\,A, B) \cong \widehat{\square}(A, \sqrt{B})$, natural in A.

Given Γ and $A : \Gamma \to \mathsf{Set}$ in $\widehat{\square}$, from Definition 2.2 we have that fibration structures $1 \to \mathsf{isFib}\,\Gamma\,A$ correspond to sections of $\mathsf{fst} : (\Sigma\,p : \wp\,\Gamma, \mathsf{C}(A \circ p)) \to \wp\,\Gamma$ and hence, transposing across the adjunction $\wp \dashv \sqrt{\ }$, to morphisms making the outer square commute in the right-hand diagram below:

We therefore have that fibration structures for A correspond to sections of the pullback $\pi_1 : R_\Gamma A \to \Gamma$ of $\sqrt{\mathsf{fst}}$ along the unit $\eta_\Gamma : \Gamma \to \sqrt{(\wp\,\Gamma)}$ of the adjunction at Γ (which is the adjoint transpose of $\mathsf{id} : \wp\,\Gamma \to \wp\,\Gamma$). This characterization of fibration structure does not depend on the particular definition of C, so should apply to many notions of fibration. We will show how it leads to the construction of a universe $\mathsf{U} = R_{\mathsf{Set}}\mathsf{id}$ and family $\pi_1 : R_{\mathsf{Set}}\mathsf{id} \to \mathsf{Set}$ which is a classifier for fibrations. However, there are two problems that have to be solved in order to carry out the construction within type theory:

- First, for Elcode in (11) to be an equality (rather than just an isomorphism), one needs the choice of $R_\Gamma A$ to be strictly functorial with respect to re-indexing along Γ (and hence to be a dependent right adjoint in the sense of [12]).
- Secondly, one cannot use ordinary type theory as the internal language to formulate the construction, because the right adjoint to $\wp$ does not internalize, as the following theorem shows.

▶ **Theorem 5.1.** *There is no* internal *right adjoint to the path functor* $\wp : \widehat{\square} \to \widehat{\square}$ *for cubical sets. In other words, there is no family of natural isomorphisms* $(\wp\,\_ \to B) \cong (\_ \to \sqrt{B}) : \widehat{\square} \to \widehat{\square}$ *(for $B \in \widehat{\square}$).*

---

[9] Warning: the adjective "tiny" is sometimes used to describe an object X of a $\mathcal{V}$-enriched cocomplete category $\mathcal{C}$ for which the hom $\mathcal{V}$-functor $\mathcal{C}(X, \_) : \mathcal{C} \to \mathcal{V}$ preserves colimits; see [35] for example. We prefer Kelly's term *small-projective object* for this property. In the special case that $\mathcal{V} = \mathcal{C}$ and $\mathcal{C}$ is cartesian closed and has sufficient properties for there to be an adjoint functor theorem, then a small-projective object is in particular a tiny one in the sense we use here.

$\sqrt{} : (A :: \mathsf{Set}_n) \to \mathsf{Set}_n$

$\mathsf{R} : \{A :: \mathsf{Set}_n\}\{B :: \mathsf{Set}_m\}(f :: \wp\, A \to B) \to A \to \sqrt{B}$

$\mathsf{L} : \{A :: \mathsf{Set}_n\}\{B :: \mathsf{Set}_m\}(g :: A \to \sqrt{B}) \to \wp\, A \to B$

$\mathsf{LR} : \{A :: \mathsf{Set}_n\}\{B :: \mathsf{Set}_m\}\{f :: \wp\, A \to B\} \to \mathsf{L}(\mathsf{R}\, f) \equiv f$

$\mathsf{RL} : \{A :: \mathsf{Set}_n\}\{B :: \mathsf{Set}_m\}\{g :: A \to \sqrt{B}\} \to \mathsf{R}(\mathsf{L}\, g) \equiv g$

$\mathsf{R}\wp : \{A :: \mathsf{Set}_n\}\{B :: \mathsf{Set}_m\}\{C :: \mathsf{Set}_k\}(g :: A \to B)(f :: \wp\, B \to C) \to \mathsf{R}(f \circ \wp'g) \equiv \mathsf{R}f \circ g$

Figure 1 Axioms for tinyness of the interval in crisp type theory.

Proof. It is an elementary fact about adjoint functors that such a family of natural isomorphisms is also natural in B. Note that $\wp\top \cong \top$. So if we had such a family, then we would also have isomorphisms $B \cong (\top \to B) \cong (\wp\top \to B) \cong (\top \to \sqrt{B}) \cong \sqrt{B}$ which are natural in B. Therefore $\sqrt{}$ would be isomorphic to the identity functor and hence so would be its left adjoint $\wp$. Hence $\mathbb{I} \to \_$ and $\top \to \_$ would be isomorphic functors $\widehat{\square} \to \widehat{\square}$, which implies (by the internal Yoneda Lemma) that $\mathbb{I}$ is isomorphic to the terminal object $\top$, contradicting the fact that $\mathbb{I}$ has two distinct global elements. ◀

We will solve the first of the two problems mentioned above in the same way that Voevodsky [41] solves a similar strictness problem (see also [12, Section 6]): apply $\sqrt{}$ once and for all to the displayed universe and then re-index, rather than *vice versa* (as done above). The second problem is solved by using the crisp type theory of the previous section to make the right adjoint $\sqrt{}$ suitably global. The axioms we use are given in Fig. 1. The function R gives the operation for transposing (global) morphisms across the adjunction $\wp \dashv \sqrt{}$, with inverse L (the bijection being given by RL and LR); and $\mathsf{R}\wp$ is the naturality of this operation. The other properties of an adjunction follow from these, in particular its functorial action $\sqrt{}' : \{A :: \mathsf{Set}_n\}\{B :: \mathsf{Set}_m\}(f :: A \to B) \to \sqrt{A} \to \sqrt{B}$. Note that Fig. 1 assumes that the right adjoint to $\mathbb{I} \to (\_)$ preserves universe levels. The soundness of this for $\widehat{\square}$ relies on the fact that this adjoint is given by right Kan extension [26, Chap. X] along $\_ \times \mathsf{I} : \square \to \square$ and hence sends a presheaf valued in the nth Grothendieck universe to another such.

▶ **Theorem 5.2** (Universe construction[10])**.** *For fibrations as in Definition 2.2 with any definition of composition structure* C *(e.g. the CCHM one in* (10)*), assuming axioms* (1)–(4) *and a tiny (Fig. 1) interval, there is a universe* U *equipped with a fibration* El *which is* classifying *in the sense that we have*

$$
\begin{aligned}
&\mathsf{U} : \mathsf{Set}_2 \\
&\mathsf{El} : \mathsf{Fib}_0\, \mathsf{U} \\
&\mathsf{code} : \{\Gamma :: \mathsf{Set}\}(\Phi :: \mathsf{Fib}_0\, \Gamma) \to \Gamma \to \mathsf{U} \\
&\mathsf{Elcode} : \{\Gamma :: \mathsf{Set}\}(\Phi :: \mathsf{Fib}_0\, \Gamma) \to \mathsf{El}[\mathsf{code}\,\Phi] \equiv \Phi \\
&\mathsf{codeEl} : \{\Gamma :: \mathsf{Set}\}(\gamma :: \Gamma \to \mathsf{U}) \to \mathsf{code}(\mathsf{El}[\gamma]) \equiv \gamma
\end{aligned}
\tag{16}
$$

---

[10] We just construct a universe for fibrations with fibers in Set_0; similar universes $\mathsf{U}_n : \mathsf{Set}_{2\sqcup n}$ can be constructed for fibrations with fibers in Set_n, for each n; see `theorem-5-2.agda` at `https://doi.org/10.17863/CAM.22369`.

Proof. Consider the display function associated with the first universe:

$$\mathsf{Elt}_1 : \mathsf{Set}_2 \qquad\qquad\qquad\qquad \mathsf{pr}_1 : \mathsf{Elt}_1 \to \mathsf{Set}_1 \qquad\qquad (17)$$
$$\mathsf{Elt}_1 = \Sigma\, A : \mathsf{Set}_1\,,\, A \qquad\qquad\qquad \mathsf{pr}_1(A, x) = A$$

We have $\mathsf{C} : \wp\,\mathsf{Set}_0 \to \mathsf{Set}_1$ and hence using the transpose operation from Fig. 1, $\mathsf{R\,C} : \mathsf{Set}_0 \to \sqrt{}\,\mathsf{Set}_1$. We define $\mathsf{U} : \mathsf{Set}_2$ by taking a pullback:

$$\mathsf{U} = \Sigma\, A : \mathsf{Set}\,,\, (\Sigma\, B : \sqrt{}\,\mathsf{Elt}_1\,,\, (\sqrt{}\,\mathsf{'pr}_1\, B \equiv \mathsf{R\,C}\, A)) \qquad (18)$$
$$\pi_1(A\,,\, (\_\,,\,\_)) = A$$
$$\pi_2(\_\,,\, (B\,,\,\_)) = B$$

Transposing this square across the adjunction $\wp \dashv \sqrt{}$ gives $\mathsf{pr}_1 \circ \mathsf{L}\,\pi_2 = \mathsf{C} \circ \wp\,{}'\pi_1 : \wp\,\mathsf{U} \to \mathsf{Set}_1$. Considering the first and second components of $\mathsf{L}\,\pi_2$, we have $\mathsf{L}\,\pi_2 \equiv \langle \mathsf{C} \circ \wp\,{}'\pi_1\,,\, \upsilon \rangle$ for some $\upsilon : (p : \wp\,\mathsf{U}) \to \mathsf{C}(\wp\,{}'\pi_1\, p)$; hence υ is an element of $\mathsf{isFib}\,\mathsf{U}\,\pi_1$ and so we can define

$$\mathsf{El} : \mathsf{Fib}_0\,\mathsf{U} \qquad \mathsf{El} = (\pi_1, \upsilon) \qquad\qquad (19)$$

So it just remains to construct the functions in (16). Given $\Gamma :: \mathsf{Set}$ and $\Phi = (A, \alpha) :: \mathsf{Fib}_0\,\Gamma$, we have $\alpha :: \mathsf{isFib}\,\Gamma\, A = (p : \wp\,\Gamma) \to \mathsf{C}(A \circ p)$. So the outer square in the left-hand diagram below commutes:

Transposing across the adjunction $\wp \dashv \sqrt{}$, this means that the outer square in the right-hand diagram also commutes and therefore induces a function $\mathsf{code}\,\Phi : \Gamma \to \mathsf{U}$ to the pullback. So there are proofs of $\pi_1 \circ \mathsf{code}\,\Phi \equiv A$ and $\pi_2 \circ \mathsf{code}\,\Phi \equiv \mathsf{R}\,\langle \mathsf{C} \circ \wp\,{}'A\,,\, \alpha \rangle$. Transposing the latter back across the adjunction gives a proof of $\mathsf{L}\,\pi_2 \circ \wp\,{}'(\mathsf{code}\,\Phi) \equiv \langle \mathsf{C} \circ \wp\,{}'A\,,\, \alpha \rangle$; and since $\mathsf{L}\,\pi_2 \equiv \langle \mathsf{C} \circ \wp\,{}'\pi_1\,,\, \upsilon \rangle$, this in turn gives a proof of $\upsilon \circ \wp\,{}'(\mathsf{code}\,\Phi) \equiv \alpha$. Combining this with the proof of $\pi_1 \circ \mathsf{code}\,\Phi \equiv A$, we get the desired element $\mathsf{Elcode}\,\Phi$ of $\mathsf{El}[\mathsf{code}\,\Phi] = (\pi_1 \circ \mathsf{code}\,\Phi\,,\, \upsilon \circ \mathsf{code}\,\Phi) \equiv (A, \alpha) = \Phi$. Finally, taking $\Gamma = \mathsf{U}$ and $\Phi = \mathsf{El}$ in (20), the uniqueness property of the pullback implies that $\mathsf{code}\,\mathsf{El} \equiv \mathsf{id}$; and similarly, for any $\gamma :: \Delta \to \Gamma$ we have that $(\mathsf{code}\,\Phi) \circ \gamma \equiv \mathsf{code}(\Phi[\gamma])$. Together these properties give us the desired element codeEl of $\mathsf{code}(\mathsf{El}[\gamma]) \equiv (\mathsf{code}\,\mathsf{El}) \circ \gamma \equiv \mathsf{id} \circ \gamma = \gamma$. ◀

▶ Remark 5.3. The above theorem can be generalized by replacing the particular universe $\mathsf{id} : \mathsf{Set} \to \mathsf{Set}$ by an arbitrary one $E_0 : U_0 \to \mathsf{Set}$. So long as the composition structure C lands in U_0, one can use the above method to construct a universe of fibrant types from among the U_0 types.[11] The application of this generalization we have in mind is to directed type theory; for example one can first construct the universe of fibrant types in the CCHM sense and then make a universe of covariant discrete fibrations in the Riehl-Shulman [34] sense from the fibrant types (repeating the construction with a different interval object).

---

[11] See `theorem-5-2-relative.agda` at `https://doi.org/10.17863/CAM.22369`.

$$\_ \sqcap \_ : \mathbb{I} \to \mathbb{I} \to \mathbb{I} \quad 0\sqcap : (i : \mathbb{I}) \to 0 \sqcap i \equiv 0 \qquad \sqcap 0 : (i : \mathbb{I}) \to i \sqcap 0 \equiv 0$$
$$\mathrm{I}\sqcap : (i : \mathbb{I}) \to \mathrm{I} \sqcap i \equiv i \qquad \sqcap\mathrm{I} : (i : \mathbb{I}) \to i \sqcap \mathrm{I} \equiv i$$
$$\mathsf{rev} : \mathbb{I} \to \mathbb{I} \qquad \mathsf{revrev} : (i : \mathbb{I}) \to \mathsf{rev}\,(\mathsf{rev}\,i) \equiv i \quad \mathsf{rev0} : \mathsf{rev}\,0 \equiv \mathrm{I}$$
$$\mathsf{isPropcof} : (\varphi : \mathsf{Set})(u\,v : \mathsf{cof}\,\varphi) \to u \equiv v$$
$$\mathsf{cofisProp} : (\varphi : \mathsf{Set})(\_ : \mathsf{cof}\,\varphi)(x\,y : \varphi) \to x \equiv y$$
$$\mathsf{cofExt} : (\varphi\,\psi : \mathsf{Set})(\_ : \mathsf{cof}\,\varphi)(\_ : \mathsf{cof}\,\psi)(\_ : \varphi \to \psi)(\_ : \psi \to \varphi) \to \varphi \equiv \psi$$
$$\mathsf{cof0} : (i : \mathbb{I}) \to \mathsf{cof}\,(0 \equiv i)$$
$$\mathsf{cofI} : (i : \mathbb{I}) \to \mathsf{cof}\,(\mathrm{I} \equiv i)$$
$$\mathsf{cofOr} : (\varphi\,\psi : \mathsf{Set})(\_ : \mathsf{cof}\,\varphi)(\_ : \mathsf{cof}\,\psi) \to \mathsf{cof}\,(\varphi \vee \psi)$$
$$\mathsf{cofAnd} : (\varphi\,\psi : \mathsf{Set})(\_ : \mathsf{cof}\,\varphi)(\_ : \varphi \to \mathsf{cof}\,\psi) \to \mathsf{cof}\,(\varphi \times \psi)$$
$$\mathsf{cof\forall\mathbb{I}} : (\varphi : \mathbb{I} \to \mathsf{Set})(\_ : (i : \mathbb{I}) \to \mathsf{cof}\,(\varphi\,i)) \to \mathsf{cof}\,((i : \mathbb{I}) \to \varphi\,i)$$
$$\mathsf{strax} : (\varphi{:}\mathsf{Set})(\_{:}\mathsf{cof}\,\varphi)(A : \mathsf{Set}_n)(t : \varphi \to (\Sigma\,B : \mathsf{Set}_n\,,\,A \cong B)) \to$$
$$\Sigma\,T : (\Sigma\,B : \mathsf{Set}_n\,,\,A \cong B)\,,\,t \nearrow T$$

🟨 **Figure 2** Further axioms needed for the CCHM model.

▶ Remark 5.4. The results in this section only make use of the fact that the functor $\sqrt{} : \widehat{\square} \to \widehat{\square}$ is right adjoint to the exponential $\mathbb{I} \to (\_)$ and we saw at the beginning of this section why such a right adjoint exists. It is possible to give an explicit description of presheaves of the form $\sqrt{\Gamma}$, but so far we have not found such a description to be useful.

6 Applications

Models. Theorem 5.2 is the missing piece that allows a completely internal development of a model of univalent foundations based upon the CCHM notion of fibration, albeit internal to crisp type theory rather than ordinary type theory. One can define a CwF in crisp type theory whose objects are crisp types $\Gamma :: \mathsf{Set}_2$, whose morphisms are crisp functions $\gamma :: \Gamma' \to \Gamma$, whose families are crisp CCHM fibrations $\Phi = (A, \alpha) :: \mathsf{Fib}_0\,\Gamma$ and whose elements are crisp dependent functions $f :: (x : \Gamma) \to A\,x$. To see that this gives a model of univalent foundations one needs to prove:
(a) The CwF is a model of intensional type theory with Π-types and inductive types (Σ-types, identity types, booleans, W-types, . . .).
(b) The type $\mathsf{U} :: \mathsf{Set}_2$ constructed in Theorem 5.2 is fibrant (as a family over the unit type).
(c) The classifying fibration $\Phi :: \mathsf{Fib}_0\,\mathsf{U}$ satisfies the univalence axiom in this CwF.

Although we have yet to complete the formal development in Agda-flat, these should be provable from axioms (1)–(4) and Fig. 1, together with some further assumptions about the interval object and cofibrant types listed in Fig. 2. Part (a) was carried out in prior work, albeit in the setting with an impredicative universe of propositions [32]. In the predicative version considered here, we replace the impredicative universe of propositions with axioms asserting that being cofibrant is a mere proposition ($\mathsf{isPropcof}$), that cofibrant types are mere propositions ($\mathsf{cofisProp}$) and satisfy propositional extensionality (cofExt). These axioms are satisfied by $\widehat{\square}$ provided we interpret $\mathsf{cof} : \mathsf{Set} \to \mathsf{Set}$ as $\mathsf{cof}\,A = \exists \varphi : \Omega\,,\,\varphi \in \mathsf{Cof} \wedge A \equiv \{\_ : \top \mid \varphi\}$, using the subobject $\mathsf{Cof} \rightarrowtail \Omega$ corresponding to the face lattice in [13] (see [32, Definition 8.6]). Axioms $\mathsf{cof0}$, cofI, cofOr, cofAnd, $\mathsf{cof\forall\mathbb{I}}$ and strax correspond to the axioms ax_5–ax_9 from [32]; in strax, $\cong$ is the usual internal statement of isomorphism. cofAnd is the dominance axiom that guarantees that cofibrations compose. Note that axiom

cofOr uses an operation sending mere propositions φ and ψ to the mere proposition $\varphi \vee \psi$ that is the propositional truncation of their disjoint union; the existence of this operation either has to be postulated, or one can add axioms for *quotient types* [18, Section 3.2.6.1] to crisp type theory, (of which propositional truncation is an instance), in which case function extensionality (1) is no longer needed as an axiom, since it is provable using quotient types [39, Section 6.3]. Since in this paper we have taken a CCHM fibration to just give a composition operation for cofibrant partial paths from 0 to I and not vice versa, in Fig. 2 we have postulated a path-reversal operation rev; this and the other axioms for $\mathbb{I}$ in that figure suffice to give a "connection algebra" structure on $\mathbb{I}$ [32, axioms ax_3 and ax_4].

Part (b) can be proved using a version of the *glueing* operation from [13], which is definable within crisp type theory as in [32, Section 6] and [10, Section 4.3.2]. The strictness axiom strax in Fig. 2 is needed to define this; and the assumption that cofibrant types are closed under $\mathbb{I}$-indexed $\forall$ (cof$\forall\mathbb{I}$) is used to define the appropriate fibration structure for glueing.

Part (c) can be proved as in [31, Section 6] using a characterization of univalence somewhat simpler than the original definition of Voevodsky [39, Section 2.10]. The axiom strax gets used to turn isomorphisms into paths; and the axiom cof$\forall\mathbb{I}$ is used to "realign" fibration structures that agree on their underlying types (see [31, Lemma 6.2]).

▶ Remark 6.1 (The interval is connected). Fig. 2 does not include an axiom asserting that the interval is connected, because that is implied by its tinyness (Fig. 1). Connectedness was postulated as ax_1 in [32] and used to prove that CCHM fibrations are closed under inductive type formers (and in particular that the natural number object is fibrant). The proof [32, Thm 8.2] that the interval in cubical sets is connected essentially uses the fact that $\widehat{\square}$ is a cohesive topos (Remark 4.1). However it also follows directly from the tinyness property: connectedness holds iff $(\mathbb{I} \to \mathbb{B}) \cong \mathbb{B}$, where $\mathbb{B} = \top + \top$ is the type of Booleans. Since we postulate that $\mathbb{I} \to \_$ has a right adjoint, it preserves this coproduct and hence $(\mathbb{I} \to \mathbb{B}) \cong (\mathbb{I} \to \top) + (\mathbb{I} \to \top) \cong \top + \top = \mathbb{B}$.

▶ Remark 6.2 (Alternative models). We have focussed on axioms satisfied by $\widehat{\square}$ and the CCHM notion of fibration in that presheaf topos. However, the universe construction in Theorem 5.2 also applies to the cartesian cubical set models [5], and we expect it is possible to give proofs in crisp type theory of its fibrancy and univalence as well.

In this paper we only consider "cartesian" path-based models of type theory, in which a path is an arbitrary function out of an interval object, or in other words, the path functor is given by an exponential. The models in [22] and [9] are not cartesian in that sense—the path functors they use are right adjoint to certain functorial cylinders [17] not given by cartesian product.[12] However, those path functors do have right adjoints (given by right Kan extension to suitable "shift" functors on the domain category of the presheaf toposes involved) and universes in these models can be constructed using the method of Theorem 5.2. (Our Agda proof of that theorem does not depend upon the path functor being an actual exponential.) A proof in crisp type theory that those universes are fibrant and univalent may require a modification of our axiomatic treatment of cofibrancy; we leave this for future work.

Universe hierarchies.   Given that there are many notions of fibration that one may be interested in, it is natural to ask how relationships between them induce relationships between universes of fibrant types. As motivating examples of this, we might want a cubical

---

[12] Furthermore, obvious candidates for an interval object are not necessarily tiny in those models—for example, for the 1-simplex $\Delta[1]$ the exponential $\Delta[1] \to (\_)$ in the topos $\widehat{\Delta}$ of simplicial sets does not have a right adjoint; thanks to a referee for pointing this out.

type theory with a universe of fibrations with *regularity*, an extra strictness corresponding to the computation rule for identity types in intensional type theory; or a three-level directed type theory with non-fibrant, fibrant, and co/contravariant universes. Towards building such hierarchies, in the companion code[13] we have shown in crisp type theory that universes are functorial in the notion of fibration they encapsulate: when one notion of fibrancy implies another, the first universe includes the second.

▶ **Proposition 6.3.** *Let* $\mathsf{C}^1, \mathsf{C}^2 : \wp\,\mathsf{Set}_n \to \mathsf{Set}_{1 \sqcup n}$ *be two notions of composition,* isFib^1 *and* isFib^2 *the corresponding fibration structures, and* U^1 *and* U^2 *the corresponding classifying universes. A morphism of fibration structures is a function* $f_{\Gamma,A} : \mathsf{isFib}^1\,\Gamma\,A \to \mathsf{isFib}^2\,\Gamma\,A$ *for all* Γ *and* A*, such that* f *is stable under reindexing (given* $h : \Delta \to \Gamma$*, and* $\phi : \mathsf{isFib}^1\,\Gamma\,A$*,* $f_{\Gamma,A}(\phi)\circ(\wp'h) \equiv f_{\Delta,A\circ f}(\phi[h])$*). Then a morphism of fibrations* f *induces a function* $\mathsf{U}^1 \to \mathsf{U}^2$*, and this preserves identity and composition.* ◀

7 Conclusion

Since the appearance of the CCHM [13] constructive model of univalence, there has been a lot of work aimed at analysing what makes this model tick, with a view to simplifying and generalizing it. Some of that work, for example by Gambino and Sattler [17, 35], uses category theory directly, and in particular techniques associated with the notion of Quillen model structure. Here we have continued to pursue the approach that uses a form of type theory as an internal language in which to describe the constructions associated with this model of univalent foundations [32, 10]. For those familiar with the language of type theory, we believe this provides an appealingly simple and accessible description of the notion of fibration and its properties in the CCHM model and in related models. We recalled why there can be no internal description of the univalent universe itself if one uses ordinary type theory as the internal language. Instead we extended ordinary type theory with a suitable modality and then gave a universe construction that hinges upon the tinyness property enjoyed by the interval in cubical sets. We call this language *crisp type theory* and our work inside it has been carried out and checked using an experimental version of Agda provided by Vezzosi [2].

───── **References** ─────

1 P. Aczel. On relating type theories and set theories. In *Proc. TYPES 1998*, volume 1657 of *Lecture Notes in Computer Science*, pages 1–18, 1999.

2 Agda-flat. URL: `https://github.com/agda/agda/tree/flat`.

3 Agda Project. URL: `wiki.portal.chalmers.se/agda`.

4 T. Altenkirch, P. Capriotti, and N. Kraus. Extending homotopy type theory with strict equality, 2016. `arXiv:1604.03799`.

5 C. Angiuli, G. Brunerie, T. Coquand, K.-B. Hou (Favonia), R. Harper, and D. R. Licata. Cartesian cubical type theory, 2017. URL: `https://github.com/dlicata335/cart-cube/blob/master/cart-cube.pdf`.

6 C. Angiuli, K.-B. Hou (Favonia), and R. Harper. Computational higher type theory III: univalent universes and exact equality, 2017. `arXiv:1712.01800`.

7 S. Awodey. Notes on cubical models of type theory, 2016. URL: `https://github.com/awodey/math/blob/master/Cubical/cubical.pdf`.

8 R. Balbes and P. Dwinger. *Distributive Lattices*. University of Missouri Press, 1975.

---

[13] see `proposition-6-2.agda` at `https://doi.org/10.17863/CAM.22369`

9   M. Bezem, T. Coquand, and S. Huber. A model of type theory in cubical sets. In *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, volume 26 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 107–128, 2014.

10   L. Birkedal, A. Bizjak, R. Clouston, H. B. Grathwohl, B. Spitters, and A. Vezzosi. Guarded cubical type theory. *Journal of Automated Reasoning*, 2018.

11   U. Buchholtz and E. Morehouse. Varieties of cubical sets. In *Relational and Algebraic Methods in Computer Science: 16th International Conference (RAMiCS 2017), Proceedings*, 2017.

12   R. Clouston, B. Mannaa, R. E. Møgelberg, A. M. Pitts, and B. Spitters. Modal dependent type theory and dependent right adjoints. arXiv:1804.05236, 2018.

13   C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. In T. Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:34, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

14   V. de Paiva and E. Ritter. Fibrational modal type theory. *Electronic Notes in Theoretical Computer Science*, 323:143–161, 2016. Proceedings of the Tenth Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2015).

15   P. Dybjer. Internal type theory. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs*, volume 1158 of *Lecture Notes in Computer Science*, pages 120–134. Springer Berlin Heidelberg, 1996.

16   D. Frumin and B. Van Den Berg. A homotopy-theoretic model of function extensionality in the effective topos. *Mathematical Structures in Computer Science*, 2018. To appear.

17   N. Gambino and C. Sattler. The Frobenius condition, right properness, and uniform fibrations. *Journal of Pure and Applied Algebra*, 221:3027–3068, 2017.

18   M. Hofmann. *Extensional Concepts in Intensional Type Theory*. PhD thesis, University of Edinburgh, 1995.

19   M. Hofmann. Syntax and semantics of dependent types. In A. M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 79–130. Cambridge University Press, 1997.

20   M. Hofmann and T. Streicher. Lifting Grothendieck universes. Unpublished note, 1999.

21   P. T. Johnstone. *Sketches of an Elephant, A Topos Theory Compendium, Volumes 1 and 2*. Number 43–44 in Oxford Logic Guides. Oxford University Press, 2002.

22   C. Kapulkin and P. L. Lumsdaine. The simplicial model of univalent foundations (after Voedodsky). arXiv:1211.2851v4, jun 2016.

23   F. W. Lawvere. Toward the description in a smooth topos of the dynamically possible motions and deformations of a continuous body. *Cahiers de Topologie et Géométrie Différentielle Catégoriques*, 21(4):377–392, 1980.

24   F. W. Lawvere. Axiomatic cohesion. *Theory and Applications of Categories*, 19(3):41–49, 2007.

25   Z. Luo. Notes on universes in type theory. Lecture notes for a talk at the Institute for Advanced Study, Princeton, 2012.

26   S. MacLane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics 5. Springer, 1971.

27   P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

28   A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Trans. Comput. Logic*, 9(3):23:1–23:49, 2008.

29   A. Nuyts, A. Vezzosi, and D. Devriese. Parametric quantifiers for dependent type theory. *Proc. ACM Program. Lang.*, 1(ICFP):32:1–32:29, aug 2017.

30 I. Orton and A. M. Pitts. Axioms for modelling cubical type theory in a topos. In *Proc. CSL 2016*, volume 62 of *LIPIcs*, pages 24:1–24:19, 2016.

31 I. Orton and A. M. Pitts. Decomposing the univalence axiom. arXiv:1712.04890, dec 2017.

32 I. Orton and A. M. Pitts. Axioms for modelling cubical type theory in a topos. *Logical Methods in Computer Science*, 2018. Special issue for CSL 2016, to appear; arXiv:1712.04864. Revised and expanded version of [30].

33 F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001.

34 E. Riehl and M. Shulman. A type theory for synthetic ∞-categories. arXiv:1705.07442, dec 2017.

35 C. Sattler. The equivalence extension property and model structures. arXiv:1704.06911, 2017.

36 M . Shulman. Brouwer's fixed-point theorem in real-cohesive homotopy type theory. *Mathematical Structures in Computer Science*, pages 1–86, 2017.

37 B. Spitters. Cubical sets and the topological topos. arXiv:1610.05270, 2016.

38 T. Uemura. Cubical assemblies and independence of the propositional resizing axiom. arXiv:1803.06649, apr 2018.

39 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations for Mathematics*. `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

40 V. Voevodsky. A simple type system with two identity types, 2013. URL: `https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/HTS.pdf`.

41 V. Voevodsky. C-systems defined by universe categories: Presheaves. arXiv:1706.03620, jun 2017.

42 D. Yetter. On right adjoints to exponentible functors. *Journal of Pure and Applied Algebra*, 45:287–304, 1987. Corrections in volume 58:103–105, 1989.

The Clocks They Are Adjunctions
Denotational Semantics for Clocked Type Theory

Bassel Mannaa
Department of Computer Science, IT University of Copenhagen, Copenhagen, Denmark
basm@itu.dk

Rasmus Ejlers Møgelberg
Department of Computer Science, IT University of Copenhagen, Copenhagen, Denmark
mogel@itu.dk

―――― **Abstract** ――

Clocked Type Theory (CloTT) is a type theory for guarded recursion useful for programming with coinductive types, allowing productivity to be encoded in types, and for reasoning about advanced programming language features using an abstract form of step-indexing. CloTT has previously been shown to enjoy a number of syntactic properties including strong normalisation, canonicity and decidability of type checking. In this paper we present a denotational semantics for CloTT useful, e.g., for studying future extensions of CloTT with constructions such as path types.

The main challenge for constructing this model is to model the notion of ticks used in CloTT for coinductive reasoning about coinductive types. We build on a category previously used to model guarded recursion, but in this category there is no object of ticks, so tick-assumptions in a context can not be modelled using standard tools. Instead we show how ticks can be modelled using adjoint functors, and how to model the tick constant using a semantic substitution.

1 Introduction

In recent years a number of extensions of Martin-Löf type theory [14] have been proposed to enhance the expressiveness or usability of the type theory. The most famous of these is Homotopy Type Theory [18], but other directions include the related Cubical Type Theory [11], FreshMLTT [17], a type theory with name abstraction based on nominal sets, and Type Theory in Color [4] for internalising relational parametricity in type theory. Many of these extensions use denotational semantics to argue for consistency and to inspire constructions in the language.

This paper is part of a project to extend type theory with guarded recursion [16], a variant of recursion that uses a modal type operator ▷ (pronounced 'later') to preserve consistency of the logical reading of type theory. The type ▷A should be read as classifying data of type A available one time step from now, and comes with a map next : $A \to \, ▷A$

3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018).
Editor: Hélène Kirchner; Article No. 23; pp. 23:1–23:17

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

and a fixed point operator mapping a function $f : \triangleright A \to A$ to a fixed point for $f \circ \mathsf{next}$. This, in combination with *guarded recursive types*, i.e., types where the recursive variable is guarded by a $\triangleright$, e.g., $\mathsf{Str} \equiv \mathbb{N} \times \triangleright\mathsf{Str}$ gives a powerful type theory in which operational models of combinations of advanced programming language features such as higher-order store [7] and nondeterminism [8] can be modelled using an abstract form of step-indexing [1]. Combining this with a notion of clocks, indexing the $\triangleright$ operator with clock names, and universal quantification over clocks, one can encode coinduction using guarded recursion, allowing productivity [12] of coinductive definitions to be encoded in types [2].

The most recent type theory with guarded recursion is Clocked Type Theory (CloTT) [3], which introduces the notion of ticks on a clock. Ticks are evidence that time has passed and can be used to unpack elements of type $\triangleright A$ to elements of A. In fact, in CloTT, $\triangleright A$ is a special form of function type from ticks to A. The combination of ticks and clocks in CloTT can be used for coinductive reasoning about coinductive types, by encoding the *delayed substitutions* of [9].

Bahr et al [3] have shown that CloTT can be given a reduction semantics satisfying strong normalisation, confluence and canonicity. This establishes that productivity can indeed be encoded in types: For a closed term t of stream type, the n'th element can be computed in finite time. These syntactic results also imply soundness of the type theory. However, these results have only been established for a core type theory without, e.g., identity types, and the arguments can be difficult to extend to larger calculi. In particular, we are interested in extending CloTT with path types as in Guarded Cubical Type Theory [5]. Therefore a denotational model of CloTT can be useful, and this paper presents such a model.

The work presented here builds on a number of existing models for guarded recursion. The most basic such, modelling the single clock case, is the topos of trees model [7], in which a closed type is modelled as a family of sets X_n indexed by natural numbers n, together with restriction maps of the form $X_{n+1} \to X_n$ for every n. In other words, a type is a presheaf over the ordered natural numbers. In this model $\triangleright$ is modelled as $(\triangleright X)_0 = 1$ and $(\triangleright X)_{n+1} = X_n$ and guarded recursion reduces to natural number recursion. The guarded recursive type Str mentioned above can be modelled in the topos of trees as $\mathsf{Str}(n) = 1 \times \mathbb{N}^n$.

Bizjak and Møgelberg [10] recently extended this model to the case of many clocks, using a category $\mathsf{Set}^{\mathbb{T}}$ of covariant presheaves over a category $\mathbb{T}$ of time objects, i.e., pairs of a finite set X and a map $X \to \mathbb{N}$. In this model, universal quantification over clocks is modelled by constructing an object in the topos of trees and taking the limit of that. For example, taking the limits over the object Str gives the usual coinductive type of streams over natural numbers.

The main challenge when adapting the model of [10] to CloTT is to model ticks, which were not present in the language modelled in [10]. In particular, how does one model tick assumptions of the form $\alpha : \kappa$ in a context, when there appears to be no object of ticks in the model to be used as the denotation of the clock κ. In this paper we observe that these assumptions can be modelled using a left adjoint $\blacktriangleleft^\kappa$ to the functor $\blacktriangleright^\kappa$ used in [10] to model $\triangleright^\kappa$ the delay modality associated to the clock κ. Precisely we model context extension as $[\![\Gamma, \alpha : \kappa]\!] = \blacktriangleleft^\kappa [\![\Gamma]\!]$. To clarify what is needed to model ticks, we focus on a fragment of CloTT called the *tick calculus* capturing just the interaction of ticks with dependent types. We show that the tick calculus can be modelled soundly in a category with family [13] (a standard notion of model for dependent type theory), with an adjunction $\mathsf{L} \dashv \mathsf{R}$ of endofunctors on the underlying category, for which the right adjoint lifts to types and terms, and there is a natural transformation from L to the identity. This appears to be a general pattern seen also in the model of fresh name abstraction of FreshMLTT [17] and dependent path types

in cubical type theory [11]. Similarly challenging is how to model the special tick constant ⋄. Since there is no object of ticks, there is no element corresponding to ⋄ either. Still, we shall see that there exists a semantic substitution of ⋄ for a tick variable that can be used to model application of terms to ⋄.

The paper is organised as follows: The tick calculus and its model theory are introduced in Section 2. Section 3 introduces CloTT, omitting guarded recursive types and universes, which we leave for future work. Section 4 presents the basics of the model, in particular the presheaf category $\mathsf{Set}^{\mathbb{T}}$ and the adjunction $\blacktriangleleft^{\kappa} \dashv \blacktriangleright^{\kappa}$. The presence of ticks in contexts leads to a non-standard notion of substitutions, and we study the syntax and semantics of these in Section 5. Sections 6 and 7 extend the model with universal quantification over clocks and ⋄, respectively. Finally, Section 8 verifies the important clock irrelevance axiom, and Section 9 concludes and discusses future work.

2   A tick calculus

Before introducing CloTT we focus on a fragment to explain the notion of ticks and how to model these. To motivate ticks, consider the notion of applicative functor from functional programming [15]: a type former ▷ with maps $A \to \triangleright A$ and $\triangleright(A \to B) \to \triangleright A \to \triangleright B$ satisfying a number of equations that we shall not recall. These maps can be used for programming with the constructor ▷, but for reasoning in a dependent type theory, one needs an extension of these to dependent function types. For example, in guarded recursion one can prove a theorem X by constructing a map $\triangleright X \to X$ and taking its fixed point in X. If the theorem is that a property holds for all elements in a type of guarded streams satisfying $\mathsf{Str} \equiv \mathbb{N} \times \triangleright\mathsf{Str}$, then X will be of the form $\prod(xs : \mathsf{Str}).P$. To apply the (essentially coinductive) assumption of type $\triangleright\prod(xs : \mathsf{Str}).P$ to the tail of a stream, which has type $\triangleright\mathsf{Str}$ we need an extension of the applicative functor action.

What should the type of such an extension be? Given $a : \triangleright A$ and $f : \triangleright(\prod(x : A).B)$ the application of f to a should be something of the form $\triangleright B[??/x]$. If we think of ▷ as a delay, intuitively a is a value of type A delayed by one time, and the ?? should be the value delivered by a one time step from now. Ticks are evidence that time has passed, and they allow us to talk about values delivered in the future.

The *tick calculus* is the extension of dependent type theory with the following four rules

$$\frac{\Gamma \vdash}{\Gamma, \alpha{:}\mathrm{tick} \vdash} \qquad \frac{\Gamma, \alpha{:}\mathrm{tick} \vdash A}{\Gamma \vdash \triangleright(\alpha{:}\mathrm{tick})A}$$

$$\frac{\Gamma, \alpha{:}\mathrm{tick} \vdash t : A}{\Gamma \vdash \lambda(\alpha{:}\mathrm{tick})t : \triangleright(\alpha{:}\mathrm{tick})A} \qquad \frac{\Gamma \vdash t : \triangleright(\alpha{:}\mathrm{tick})A}{\Gamma, \beta{:}\mathrm{tick}, \Gamma' \vdash t\,[\beta] : A[\beta/\alpha]}$$

An assumption of the form $\alpha{:}\mathrm{tick}$ in a context is an assumption that one time step has passed, and α is the evidence of this. Variables on the right-hand side of such an assumption should be thought of as arriving one time step later than those on the left. Ticks can be abstracted in terms and types, so that the type constructor ▷ now comes with evidence that time has passed that can be used in its scope. The type $\triangleright(\alpha{:}\mathrm{tick})A$ can be thought of as a form of dependent function type over ticks, which we abbreviate to $\triangleright A$ if α does not occur free in A. The elimination rule states that if a term t can be typed as $\triangleright(\alpha{:}\mathrm{tick})A$ before the arrival of tick β, t can be opened using β to give something of type $A[\beta/\alpha]$. Note that the causality restriction in the typing rule prevents a term like $\lambda x.\lambda(\alpha{:}\mathrm{tick}).x\,[\alpha]\,[\alpha] : \triangleright\triangleright A \to \triangleright A$ being well typed; a tick can only be used to unpack the same term once. The context Γ'

in the elimination rule ensures that typing rules are closed under weakening, also for ticks. Note that the clock object tick is not a type.

The equality theory is likewise extended with the usual β and η rules:

$$\lambda(\alpha{:}\mathrm{tick})t\,[\beta] = t[\beta/\alpha] \qquad\qquad\qquad \lambda(\alpha{:}\mathrm{tick})(t\,[\alpha]) = t$$

As stated, the tick calculus should be understood as an extension of standard dependent type theory. In particular one can add dependent sums and function types with standard rules. Variables can be introduced from anywhere in the context, also past ticks.

We can now type the dependent applicative structure as

$$\lambda(x{:}A)\lambda(\alpha{:}\mathrm{tick})x : A \to \rhd A$$
$$\lambda f\lambda y\lambda(\alpha{:}\mathrm{tick})f\,[\alpha](y\,[\alpha]) : \rhd\left(\textstyle\prod(x:A)\,.B\right) \to \textstyle\prod(y:\rhd A)\,.\rhd(\alpha{:}\mathrm{tick}).B[y\,[\alpha]/x]$$

For a small example on how ticks in combination with the fixed point operator dfix : $(\rhd X \to X) \to \rhd X$ can be used to reason about guarded recursive data, let $\mathsf{Str} \equiv \mathbb{N} \times \rhd\mathsf{Str}$ be the type of guarded recursive streams mentioned above, and suppose $x{:}\mathsf{N} \vdash P(x)$ is a family to be thought of as a predicate on N (where $x :: xs$ is the pairing of x and xs). A lifting of P to streams would be another guarded recursive type $y{:}\mathsf{Str} \vdash \hat{P}(y)$ satisfying $\hat{P}(x :: xs) \equiv P(x) \times \rhd(\alpha{:}\mathrm{tick})\hat{P}(xs\,[\alpha])$. If $p : \Pi(x{:}\mathsf{N})P(x)$ is a proof of P we would expect that also $\Pi(y{:}\mathsf{Str})\hat{P}(y)$ can be proved, and indeed this can be done as follows. Consider first

$$f : \rhd(\Pi(y{:}\mathsf{Str})\hat{P}(y)) \to \Pi(y{:}\mathsf{Str})\hat{P}(y)$$
$$f\,q\,(x :: xs) \overset{\mathsf{def}}{=} p(x) :: \lambda(\alpha{:}\mathrm{tick})q\,[\alpha](xs\,[\alpha])$$

Then $f(\mathsf{dfix}(f))$ has the desired type.

More generally, ticks can be used to encode [3] the *delayed substitutions* of [9], which have been used to reason coinductively about coinductive data. For more examples of reasoning using these see [9]. For reasons of space, we will not model general guarded recursive types in this paper, but see Section 4 for how to model the types used above.

2.1 Modelling ticks using adjunctions

We now describe a notion of model for the tick calculus. It is based on the notion of category with families (CwF) [13], which is a standard notion of model of dependent type theory. Recall that a CwF is a pair $(\mathscr{C}, T)$ such that $\mathscr{C}$ is a category with a distinguished terminal object and $T : \mathscr{C}^{\mathrm{op}} \to \mathsf{Fam}(\mathsf{Set})$ is a functor together with a comprehension map to be recalled below. The functor T associates to every object Γ in $\mathscr{C}$ a map $T(\Gamma) : \mathsf{Tm}(\Gamma) \to \mathsf{Ty}(\Gamma)$ and to every morphism $\gamma : \Delta \to \Gamma$ maps $\mathsf{Ty}(\gamma) : \mathsf{Ty}(\Gamma) \to \mathsf{Ty}(\Delta)$ and $\mathsf{Tm}(\gamma) : \mathsf{Tm}(\Gamma) \to \mathsf{Tm}(\Delta)$ such that $T(\Delta) \circ \mathsf{Tm}(\gamma) = \mathsf{Ty}(\gamma) \circ T(\Gamma)$. Following standard conventions, we write $\Gamma \vdash A$ to mean $A \in \mathsf{Ty}(\Gamma)$ and $\Gamma \vdash t : A$ to mean $t \in T(\Gamma)^{-1}(A)$, and we write $\Delta \vdash A[\gamma]$ for $\mathsf{Ty}(\gamma)(A)$ when $\Gamma \vdash A$, and likewise $\Delta \vdash t[\gamma] : A[\gamma]$ for $\mathsf{Tm}(\gamma)(t)$ when $\Gamma \vdash t : A$. We refer to the objects of $\mathscr{C}$ as contexts, morphisms as substitutions, elements of $\mathsf{Ty}(\Gamma)$ as types and elements of $\mathsf{Tm}(\Gamma)$ as terms.

Comprehension associates to each $\Gamma \vdash A$ a context $\Gamma.A$, a substitution $\mathsf{p}_A : \Gamma.A \to \Gamma$ and a term $\Gamma.A \vdash \mathsf{q}_A : A[\mathsf{p}_A]$, such that for every $\gamma : \Delta \to \Gamma$, and $\Delta \vdash t : A[\gamma]$ there exists a unique substitution $\langle\gamma, t\rangle : \Delta \to \Gamma.A$ such that $\mathsf{p}_A \circ \langle\gamma, t\rangle = \gamma$ and $\mathsf{q}_A[\langle\gamma, t\rangle] = t$.

To model the tick calculus we need an operation L modelling the extension of a context with a tick, plus an operation R modelling $\rhd$. In the simply typed setting, R would be a right adjoint to context extension, but for dependent types this is not quite so, since these

operations work on different objects (contexts and types respectively). In the model we consider in this paper, the right adjoint does exist as an operation on contexts, but also extends to types and terms in the sense of the following definition.

▶ **Definition 1.** Let $(\mathscr{C}, T)$ be a CwF and let $\mathsf{R} : \mathscr{C} \to \mathscr{C}$ be a functor. An *extension of* R *to types and terms* is a pair of operations on types and term presented here in the form of rules

$$\frac{\Gamma \vdash A}{\mathsf{R}\Gamma \vdash \mathsf{R}A} \qquad \frac{\Gamma \vdash t : A}{\mathsf{R}\Gamma \vdash \mathsf{R}t : \mathsf{R}A}$$

commuting with substitutions in the sense that $(\mathsf{R}A)[\mathsf{R}\gamma] = \mathsf{R}(A[\gamma])$ and $(\mathsf{R}t)[\mathsf{R}\gamma] = \mathsf{R}(t[\gamma])$ hold for all substitutions γ, and commuting with comprehension in the sense that there exists an operation associating to each $\Gamma \vdash A$ a morphism $\zeta_{\Gamma,A} : \mathsf{R}\Gamma.\mathsf{R}A \to \mathsf{R}(\Gamma.A)$ in $\mathscr{C}$ inverse to $\langle \mathsf{R}\mathsf{p}_A, \mathsf{R}\mathsf{q}_A \rangle$. A *CwF with adjunction* is a pair of adjoint endofunctors $\mathsf{L} \dashv \mathsf{R} : \mathscr{C} \to \mathscr{C}$ with an extension of R to types and terms.

Given a CwF with adjunction, one can define an operation mapping types $\mathsf{L}\Gamma \vdash A$ to types $\Gamma \vdash \mathsf{R}_\Gamma A$ defined as $\mathsf{R}_\Gamma A = (\mathsf{R}A)[\eta]$ where η is the unit of the adjunction.

▶ **Lemma 2.** *There is a bijective correspondence between terms* $\mathsf{L}\Gamma \vdash a : A$ *and terms* $\Gamma \vdash b : \mathsf{R}_\Gamma A$ *for which we write* $\overline{(-)}$ *for both directions where* $\Gamma \vdash \overline{a} : \mathsf{R}_\Gamma A$ *is given by* $\overline{a} = (\mathsf{R}a)[\eta]$ *and* $\mathsf{L}\Gamma \vdash \overline{b} : A$ *is given by* $\overline{b} = \mathsf{q}_A[\epsilon \circ \mathsf{L}(\zeta_{\mathsf{L}\Gamma,A} \circ \langle \eta, b \rangle)]$. *Moreover, if* $\gamma : \Delta \to \Gamma$, $\mathsf{L}\Gamma \vdash a : A$ *and* $\Gamma \vdash b : \mathsf{R}_\Gamma A$ *then*

$$(\mathsf{R}_\Gamma A)[\gamma] = R_\Delta(A[\mathsf{L}\gamma]) \qquad\qquad \overline{a[\mathsf{L}\gamma]} = \overline{a}[\gamma] \qquad\qquad \overline{b[\gamma]} = \overline{b}[\mathsf{L}\gamma]$$

2.2 Interpretation

The notion of CwF with adjunction is almost sufficient for modelling the tick calculus, but to interpret tick weakening, we will assume given a natural transformation $\mathsf{p}_\mathsf{L} : \mathsf{L} \to \mathsf{id}_\mathscr{C}$. Defining

$$[\![\Gamma, \alpha : \mathrm{tick} \vdash]\!] = \mathsf{L}[\![\Gamma]\!]$$

p_L allows us to define a context projection $\mathsf{p}_{\Gamma'} : [\![\Gamma, \Gamma' \vdash]\!] \to [\![\Gamma \vdash]\!]$ by induction on Γ' using p_L in the case of tick variables. We can then define the rest of the interpretation as

$$[\![\Gamma, x : A, \Gamma' \vdash x : A]\!] = \mathsf{q}_A[\mathsf{p}_{\Gamma'}] \qquad\qquad [\![\Gamma \vdash \triangleright(\alpha{:}\mathrm{tick})A]\!] = \mathsf{R}_{[\![\Gamma]\!]}[\![A]\!]$$

$$[\![\Gamma \vdash \lambda(\alpha{:}\mathrm{tick})t]\!] = \overline{[\![t]\!]} \qquad\qquad [\![\Gamma, \alpha'{:}\mathrm{tick}, \Gamma' \vdash t\,[\alpha']]\!] = \overline{\overline{[\![t]\!]}}[\mathsf{p}_{\Gamma'}]$$

▶ **Proposition 3.** *The above interpretation of the tick calculus into a CwF with adjunction and tick weakening* p_L *is sound.*

3 Clocked Type Theory

Clocked Type Theory (CloTT) is an extension of the tick calculus with guarded recursion and multiple clocks. Rather than having a global notion of time as in the tick calculus, ticks are associated with clocks and clocks can be assumed and universally quantified. Judgements have a separate context of clock variables Δ, for example, the typing judgement has the form $\Gamma \vdash_\Delta t : A$, where Δ is a set of clock variables $\kappa_1, \ldots, \kappa_n$. The clock context can be thought of as a context of assumptions of the form $\kappa_1 : \mathrm{Clock}, \ldots, \kappa_n : \mathrm{Clock}$ that appear to the left of the assumptions of Γ, except that Clock is not a type. There are no operations for

Type formation rules

$$\frac{\Gamma, \alpha : \kappa \vdash_\Delta A \text{ type} \qquad \kappa \in \Delta}{\Gamma \vdash_\Delta \triangleright (\alpha : \kappa).A \text{ type}} \qquad\qquad \frac{\Gamma \vdash_{\Delta,\kappa} A \text{ type} \qquad \Gamma \vdash_\Delta}{\Gamma \vdash_\Delta \forall \kappa.A \text{ type}}$$

Typing rules

$$\frac{\Gamma \vdash_{\Delta,\kappa} t : A \qquad \Gamma \vdash_\Delta}{\Gamma \vdash_\Delta \Lambda \kappa.t : \forall \kappa.A} \qquad \frac{\Gamma \vdash_\Delta t : \forall \kappa.A \qquad \kappa' \in \Delta}{\Gamma \vdash_\Delta t[\kappa'] : A[\kappa'/\kappa]} \qquad \frac{\Gamma, \alpha : \kappa \vdash_\Delta t : A \qquad \kappa \in \Delta}{\Gamma \vdash_\Delta \lambda(\alpha : \kappa).t : \triangleright (\alpha : \kappa).A}$$

$$\frac{\Gamma \vdash_\Delta t : \triangleright (\alpha : \kappa).A \qquad \Gamma, \alpha' : \kappa, \Gamma' \vdash_\Delta}{\Gamma, \alpha' : \kappa, \Gamma' \vdash_\Delta t[\alpha'] : A[\alpha'/\alpha]} \qquad \frac{\Gamma \vdash_{\Delta,\kappa} t : \triangleright (\alpha : \kappa).A \qquad \Gamma \vdash_\Delta \qquad \kappa' \in \Delta}{\Gamma \vdash_\Delta (t[\kappa'/\kappa])[\diamond] : A[\kappa'/\kappa][\diamond/\alpha]}$$

$$\frac{\Gamma \vdash_\Delta t : \triangleright^\kappa A \to A}{\Gamma \vdash_\Delta \mathsf{dfix}^\kappa t : \triangleright^\kappa A}$$

Judgemental equality

$$(\Lambda \kappa.t)[\kappa'] \equiv t[\kappa/\kappa'] \qquad\qquad (\Lambda \kappa.t[\kappa]) \equiv t \qquad\qquad (\lambda(\alpha' : \kappa).t)[\alpha] \equiv t[\alpha/\alpha']$$

$$\lambda(\alpha : \kappa).(t[\alpha]) \equiv t \qquad\qquad (\mathsf{dfix}^\kappa t)[\diamond] \equiv t(\mathsf{dfix}^\kappa t)$$

🟨 **Figure 1** Selected typing and judgemental equality rules of Clocked Type Theory.

forming clocks, only clock variables. It is often convenient to have a single clock constant κ_0 and this can be added by working in a context of a single clock variable.

The rules for typing judgements and judgemental equality are given in Figure 1. These should be seen as an extension of a dependent type theory with dependent function and sum types, as well as extensional identity types. The rules for these are completely standard (ignoring the clock context), and thus are omitted from the figure. We write $\equiv$ for judgemental equality and $t =_A u$ for identity types. The model will also model the *identity reflection* rule

$$\frac{\Gamma \vdash_\Delta p : t =_A u}{\Gamma \vdash_\Delta t \equiv u : A}$$

of extensional type theory.

The guarded fixed point operator dfix is useful in combination with guarded recursive types. Suppose for example that we have a type of natural numbers $\mathbb{N}$ and a type of guarded recursive streams Str^κ satisfying $\mathsf{Str}^\kappa \equiv \mathbb{N} \times \triangleright^\kappa \mathsf{Str}^\kappa$. One can then use dfix for recursive programming with guarded streams, e.g., when defining a constant stream of zeros as $\mathsf{dfix}^\kappa(\lambda x.(0, x))$. The type of dfix ensures that only productive recursive definitions are typeable, e.g., $\mathsf{dfix}^\kappa(\lambda x.x)$ is not.

The tick constant $\diamond$ gives a way to execute a delayed computation t of type $\triangleright^\kappa A$ to compute a value of type A. In particular, if t is a fixed point, application to the tick constant unfolds the fixed point once. This explains the need to name ticks in CloTT: substitution of $\diamond$ for a tick variable α in a term allows for all fixed points applied to α in the term to be unfolded. In particular, the names of ticks are crucial for the strong normalisation result for CloTT in [3].

To ensure productivity, application of ⋄ must be restricted. In particular a term such as $\mathsf{dfix}^\kappa(\lambda x : \mathsf{Str}^\kappa.x\,[\diamond])$ should not be well typed. The typing rule for application to the tick constant ensures this by assuming that the clock κ associated to the delay is not free in the context of the term t. For example, the rule

$$\frac{\Gamma \vdash_{\Delta,\kappa} t : \triangleright (\alpha : \kappa).A \qquad \Gamma \vdash_\Delta}{\Gamma \vdash_{\Delta,\kappa} t\,[\diamond] : A\,[\diamond/\alpha]}$$

is admissible, which can be proved using weakening lemma for the clock variable context. This rule, however, is not closed under variable substitution, which is the motivation for the more general rule of Figure 1. The typing rule is a bit unusual, in that it involves substitution in the term in the conclusion. We shall see in Section 7.1 that this causes extra proof obligations for welldefinedness of the denotational semantics.

Universal quantification over clocks allow for coinductive types to be encoded using guarded recursive types [2]. For example $\mathsf{Str} \stackrel{\mathsf{def}}{=} \forall\kappa.\mathsf{Str}^\kappa$ is a coinductive type of streams. The head and tail maps $\mathsf{hd} : \mathsf{Str} \to \mathbb{N}$ and $\mathsf{tl} : \mathsf{Str} \to \mathsf{Str}$ can be defined as

$$\mathsf{hd}(xs) \stackrel{\mathsf{def}}{=} \pi_1(xs[\kappa_0]) \qquad\qquad \mathsf{tl}(xs) \stackrel{\mathsf{def}}{=} \Lambda\kappa.((\pi_2(xs[\kappa]))\,[\diamond])$$

using the clock constant κ_0.

Finally we recall the *clock irrelevance axiom*

$$\frac{\Gamma \vdash_\Delta t : \forall\kappa.A \qquad \Gamma \vdash_\Delta A \text{ type}}{\Gamma \vdash_\Delta \mathsf{cirr}^\kappa t : \forall\kappa'.\forall\kappa''.t[\kappa'] =_A t[\kappa'']} \tag{1}$$

crucial for correctness of the encoding of coinductive types [2]. Note that the hypothesis implies that κ is not free in A. This rule can be used to prove that $\forall\kappa.A$ is isomorphic to A if κ is not free in A. Likewise the *tick irrelevance axiom*

$$\frac{\Gamma \vdash_\Delta t : \triangleright^\kappa A}{\mathsf{tirr}^\kappa t : \triangleright (\alpha : \kappa).\triangleright (\alpha' : \kappa).t\,[\alpha] =_A t\,[\alpha']} \tag{2}$$

states that the identity of ticks is irrelevant for the equality theory, despite being crucial for the reduction semantics. Tick irrelevance implies fixed point unfolding

$$\frac{\Gamma \vdash_{\Delta,\kappa} f : \triangleright^\kappa A \to A \qquad \Gamma \vdash_\Delta \qquad \kappa' \in \Delta}{\Gamma \vdash_\Delta \mathsf{pfix}^{\kappa'} f[\kappa'/\kappa] : \triangleright (\alpha : \kappa).(\mathsf{dfix}^{\kappa'} f[\kappa'/\kappa])\,[\alpha] =_A (f(\mathsf{dfix}^{\kappa'} f))[\kappa'/\kappa]}$$

The type theory CloTT as defined in [3] also has guarded recursive types and a universe. We leave these for future work, see Section 9.

4 Presheaf semantics

The setting for the denotational semantics of CloTT is a category of covariant presheaves over a category $\mathbb{T}$ of time objects. This category has previously been used to give a model of GDTT [10].

We will assume given a countably infinite set CV of (semantic) clock variables, for which we use $\lambda, \lambda', \dots$ to range over. A *time object* is a pair $(\Theta; \vartheta)$ where Θ is a finite subset of CV and $\vartheta : \Theta \to \mathbb{N}$ is a map giving the number of ticks left on each clock in Θ. We will write the finite sets Θ as lists writing e.g., Θ, λ for $\Theta \cup \{\lambda\}$ and $\vartheta[\lambda \mapsto n]$ for the extension of ϑ to Θ, λ, or indeed for the update of ϑ, if ϑ is already defined on λ. A morphism $(\Theta; \vartheta) \to (\Theta'; \vartheta')$ is a

function $\tau : \Theta \to \Theta'$ such that $\vartheta' \tau \leq \vartheta$ in the pointwise order. The inequality allows for time to pass in a morphism, but morphisms can also synchronise clocks in Θ by mapping them to the same clock in Θ', or introduce new clocks if τ is not surjective. Define GR to be the category $\mathsf{Set}^{\mathbb{T}}$ of covariant presheaves on $\mathbb{T}$. The topos of trees can be seen as a restriction of this where time objects always have a single clock.

The category GR contains a special object of clocks, given by the first projection $\mathrm{Clk}(\Theta; \vartheta) = \Theta$. If Δ is a set, one can form the object Clk^{Δ} as $\mathrm{Clk}^{\Delta}(\Theta; \vartheta) = \Theta^{\Delta}$. Let $\mathbb{T}_{\Delta}$ be the category of elements of Clk^{Δ}, i.e., the objects are triples $(\Theta; \vartheta; f)$ where $(\Theta; \vartheta) \in \mathbb{T}$ and $f : \Delta \to \Theta$ and a morphism $\tau : (\Theta; \vartheta; f) \to (\Theta'; \vartheta'; f')$ is a morphism $\tau : (\Theta; \vartheta) \to (\Theta'; \vartheta')$ such that $\tau \circ f = f'$. A clock context Δ will be interpreted as Clk^{Δ} and contexts, types and terms in clock context Δ will be modelled in the category $\mathsf{GR}[\Delta] \stackrel{\mathsf{def}}{=} \mathsf{Set}^{\mathbb{T}_{\Delta}}$ of covariant presheaves over $\mathbb{T}_{\Delta}$. If F is a covariant presheaf over $\mathsf{GR}[\Delta]$ and $\tau : (\Theta; \vartheta; f) \to (\Theta'; \vartheta'; f')$ and $x \in F(\Theta; \vartheta; f)$ we will write $\tau \cdot x$ for $F(\tau)(x) \in F(\Theta'; \vartheta'; f')$.

To describe the model of CloTT, we start by fixing a clock context Δ and modelling the fragment of CloTT excluding universal quantification over clocks and the tick constant $\diamond$. The resulting fragment is a version of the tick calculus with one notion of tick for each clock κ in Δ. To model this, we need the structure of a CwF with adjunction on $\mathsf{GR}[\Delta]$ for each κ in Δ. Recall that, like any presheaf category, $\mathsf{GR}[\Delta]$ can be equipped with the structure of a CwF where contexts are objects, types in context Γ are presheaves over the elements of Γ and terms are sections. Precisely, a type over Γ is a mapping associating a set $A(\gamma)$ to each $\gamma \in \Gamma(\Theta; \vartheta; f)$ and to each $\tau : (\Theta; \vartheta; f) \to (\Theta'; \vartheta'; f')$ a mapping $\tau \cdot (-) : A(\gamma) \to A(\tau \cdot \gamma)$ such that $\mathsf{id} \cdot x = x$ and $(\rho\tau) \cdot x = \rho \cdot (\tau \cdot x)$ for all x, τ and ρ. A term is a mapping associating to each γ an element $t(\gamma) \in A(\gamma)$ such that $t(\tau \cdot \gamma) = \tau \cdot t(\gamma)$. We often make the underlying $\mathbb{T}_{\Delta}$ object explicit writing $t_{(\Theta; \vartheta; f)}(\gamma)$.

As an example of a model of a type, recall the type of guarded streams satisfying $\mathsf{Str}^{\kappa} \equiv \mathbb{N} \times \triangleright \mathsf{Str}^{\kappa}$ from Section 3. This is a closed type in a clock context Δ (assuming $\kappa \in \Delta$), and so will be interpreted as a presheaf in $\mathsf{GR}[\Delta]$ defined as $[\![\mathsf{Str}^{\kappa}]\!](\Theta; \vartheta; f) = \mathbb{N}^{\vartheta(f(\kappa))+1} \times \{*\}$. We will assume that the products in this associate to the right, so that this is the type of tuples of the form $(n_{\vartheta(f(\kappa))}, (\ldots, (n_0, *)) \ldots)$. This is needed to model the equality $\mathsf{Str}^{\kappa} \equiv \mathbb{N} \times \triangleright \mathsf{Str}^{\kappa}$, rather than just an isomorphism of types. Given a predicate $x{:}\mathsf{N} \vdash P$, we can lift it to a predicate $y : \mathsf{Str}^{\kappa} \vdash \hat{P}$ satisfying $\hat{P}(x : xs) \equiv P(x) \times \triangleright (\alpha : \kappa).\hat{P}(xs\,[\alpha])$ as in Section 2, by defining

$$[\![\hat{P}]\!]_{(\Theta; \vartheta; f)}(n_{\vartheta(f(\kappa))}, (\ldots, (n_0, *)) \ldots) = \{(x_{\vartheta(f(\kappa))}, (\ldots, (x_0, *)) \ldots) \mid \forall i.x_i \in [\![P]\!]_{(\Theta; \vartheta; f)}(n_i)\}$$

It is a simple calculation (using the definitions below) that these interpretations model the type equalities mentioned above.

4.1   Adjunction structure on $\mathsf{GR}[\Delta]$

For the adjunction, recall that in the topos of trees the functor $\blacktriangleright$ is defined as $(\blacktriangleright F)(n+1) = Fn$ and $(\blacktriangleright F)(0) = \{*\}$. This has a left adjoint $\blacktriangleleft$ defined as $(\blacktriangleleft F)n = F(n+1)$. The right adjoint generalises in a straight forward way to the multiclock setting of CloTT: If F is in $\mathsf{GR}[\Delta]$, define

$$(\blacktriangleright^{\kappa} F)(\Theta; \vartheta; f) = \begin{cases} F(\Theta; \vartheta[f(\kappa)-]; f) & \vartheta(f(\kappa)) > 0 \\ \{*\} & \text{otherwise} \end{cases}$$

where $\vartheta[f(\kappa)-](f(\kappa)) = \vartheta(f(\kappa)) - 1$ and $\vartheta[f(\kappa)-](\lambda) = \vartheta(\lambda)$ for $\lambda \neq f(\kappa)$. This is the same definition as used in the GDTT model of [10].

▶ **Lemma 4.** *The functor* $\blacktriangleright^{\kappa}$ *extends to types and terms.*

Proof. We just give the definitions. For $\gamma \in (\blacktriangleright^\kappa \llbracket \Gamma \rrbracket)(\Theta; \vartheta; f)$ define

$$(\blacktriangleright^\kappa \llbracket A \rrbracket)_{(\Theta; \vartheta; f)}(\gamma) = \begin{cases} \{*\} & \vartheta(f(\kappa)) = 0 \\ \llbracket A \rrbracket_{(\Theta; \vartheta[f(\kappa)-]; f)}(\gamma) & \text{otherwise} \end{cases}$$

The case for terms is similar.

The isomorphism $\zeta_{\Gamma, A}$ is given by $\zeta_{\Gamma, A}{}_{(\Theta; \vartheta; f)} = \begin{cases} \langle *, * \rangle \mapsto * & \vartheta(f(\kappa)) = 0 \\ \mathsf{id} & \text{otherwise} \end{cases}$ ◀

At first sight it would seem that one can define a left adjoint to the above functor given by $(\blacktriangleleft^\kappa F)(\Theta; \vartheta; f) = F(\Theta; \vartheta[f(\kappa)+]; f)$, where $\vartheta[f(\kappa)+]$ is defined similarly to $\vartheta[f(\kappa)-]$. Unfortunately, $\blacktriangleleft^\kappa F$ so described is not a presheaf because it has no well-defined action on maps since a map $\tau : (\Theta; \vartheta; f) \to (\Theta'; \vartheta'; f')$ does not necessarily induce a map $(\Theta; \vartheta[f(\kappa)+]; f) \to (\Theta'; \vartheta'[f'(\kappa)+]; f')$: If $\tau(f(\kappa)) = \tau(\lambda)$ there is no guarantee that $\vartheta'[f'(\kappa)+](\tau(\lambda)) \leq \vartheta[f(\kappa)+](\lambda)$.

To get the correct description of the left adjoint consider the set $f^{-1}(f(\kappa)) \subseteq \Delta$ of syntactic clocks synchronised with κ by f. Given a morphism $\tau : (\Theta; \vartheta; f) \to (\Theta'; \vartheta'; f')$, more clocks can be synchronised with κ by f' than f, but never fewer. If we think of time as flowing in the direction of morphisms, the left adjoint must take into account all the possible ways that κ could have been synchronised with fewer syntactic clocks "in the past". Such a past is given by a subset $X \subset f^{-1}(f(\kappa))$ such that $\kappa \in X$.

▶ **Lemma 5.** *The functor $\blacktriangleright^\kappa$ has a left adjoint $\blacktriangleleft^\kappa$ given by*

$$\blacktriangleleft^\kappa F(\Theta; \vartheta; f) = \coprod_{\kappa \in X \subset f^{-1}(f(\kappa))} F(\Theta; \vartheta; f)[X, \kappa+]$$

where $(\Theta; \vartheta; f)[X, \kappa+] = (\Theta, \#_\Theta; \vartheta[\#_\Theta \mapsto \vartheta(f(\kappa)) + 1]; f[X \mapsto \#_\Theta])$ for $\#_\Theta$ a chosen clock name fresh for Θ.

Finally, the projection $\mathsf{p}_{\blacktriangleleft^\kappa} : \blacktriangleleft^\kappa \to \mathsf{id}$ maps an element (X, γ) in $\blacktriangleleft^\kappa F(\Theta; \vartheta; f)$ to $\chi \cdot \gamma$ where

$$\chi : (\Theta; \vartheta; f)[X, \kappa+] \to (\Theta; \vartheta; f)$$

is defined as $\chi(\#_\Theta) = f(\kappa)$ and $\chi(\lambda) = \lambda$ for $\lambda \in \Theta$. Collectively, Lemmas 4 and 5 together with the projection $\mathsf{p}_{\blacktriangleleft^\kappa}$ state that for each κ, $\mathsf{GR}[\Delta]$ carries the structure of a model of the tick calculus. This is enough to model the tick abstractions and applications of CloTT.

The adjoint correspondent $\overline{\mathsf{p}_{\blacktriangleleft^\kappa}} : \mathsf{id} \to \blacktriangleright^\kappa$ to $\mathsf{p}_{\blacktriangleleft^\kappa}$ maps an element $\gamma \in F(\Theta; \vartheta; f)$ to its restriction in $F(\Theta; \vartheta[f(\kappa)-]; f)$. This is the map referred to as next in [10]. Moreover, a simple calculation shows that the interpretation of $\triangleright^\kappa A$, i.e., $\triangleright(\alpha : \kappa).A$ for α not free in A, is the same as in [10], namely

$$\llbracket \Gamma \vdash_\Delta \triangleright^\kappa A \text{ type} \rrbracket_{(\Theta; \vartheta; f)}(\gamma) = \llbracket A \rrbracket_{(\Theta; \vartheta[f(\kappa)-]; f)}(\gamma|_{(\Theta; \vartheta[f(\kappa)-]; f)})$$

We can thus define the interpretation of dfix as in [10] by induction on $\vartheta(f(\kappa))$:

$$\llbracket \mathsf{dfix}\, t \rrbracket_{(\Theta; \vartheta; f)}(\gamma) = \begin{cases} * & \vartheta(f(\kappa)) = 0 \\ \llbracket t\,(\mathsf{dfix}\, t) \rrbracket_{(\Theta; \vartheta[f(\kappa)-]; f)}(\gamma|_{(\Theta; \vartheta[f(\kappa)-]; f)}) & \text{Otherwise} \end{cases}$$

Finally we note soundness of the tick irrelevance axiom (2).

▶ **Proposition 6.** *If $\Gamma \vdash_\Delta t : \triangleright^\kappa A$ then*

$$\llbracket \Gamma, \alpha : \kappa, \alpha' : \kappa \vdash_\Delta t\,[\alpha] : A \rrbracket = \llbracket \Gamma, \alpha : \kappa, \alpha' : \kappa \vdash_\Delta t\,[\alpha'] : A \rrbracket$$

5   Substitutions

Having described the interpretation of the fragment of CloTT that lives within a fixed clock context Δ it remains to describe the interpretation of the universal quantification over clocks and of the tick constant $\diamond$. Quantification over clocks can be seen as a dependent product over a type of clocks, and should therefore be modelled as a right adjoint to weakening in the clock context. Weakening is an example of a substitution and, as we shall see, the tick constant $\diamond$ will also be modelled using a form of substitution. We therefore first study substitutions, which are non-standard in CloTT because of the two contexts, and because of the unusual typing rules for ticks.

5.1   Syntactic substitutions

A syntactic substitution from $\Gamma \vdash_\Delta$ to $\Gamma' \vdash_{\Delta'}$ is a pair (ν, σ) of a substitution ν of clocks for clocks and a substitution σ of terms for variables and ticks for ticks variables. Substitutions are formed according to the following rules.

- If $\nu : \Delta' \to \Delta$ is a map of sets, then $(\nu, \cdot) : \Gamma \vdash_\Delta \to \cdot \vdash_{\Delta'}$
- If $(\nu, \sigma) : \Gamma \vdash_\Delta \to \Gamma' \vdash_{\Delta'}$ and $\Gamma \vdash_\Delta t : A\,(\nu, \sigma)$ then $(\nu, \sigma[x \mapsto t]) : \Gamma \vdash_\Delta \to \Gamma', x : A \vdash_{\Delta'}$
- If $(\nu, \sigma) : \Gamma \vdash_\Delta \to \Gamma' \vdash_{\Delta'}$ and $\Gamma, \alpha : \nu(\kappa), \Gamma'' \vdash_\Delta$ and $\Gamma', \beta : \kappa \vdash_{\Delta'}$ are welformed then $(\nu, \sigma[\beta \mapsto \alpha]) : \Gamma, \alpha : \nu(\kappa), \Gamma'' \vdash_\Delta \to \Gamma', \beta : \kappa \vdash_{\Delta'}$
- If $(\nu, \sigma) : \Gamma \vdash_\Delta \to \Gamma' \vdash_{\Delta'}$, $\kappa \notin \Delta'$ and $\kappa' \in \Delta$, then

$$(\nu[\kappa \mapsto \nu(\kappa')], \sigma[\alpha \mapsto \diamond]) : \Gamma \vdash_\Delta \to \Gamma', \alpha : \kappa \vdash_{\Delta',\kappa}$$

Here $A\,(\nu, \sigma)$ is the result of substituting A along (ν, σ) which is defined in the standard way.

5.2   Semantic substitutions

The clock substitution ν gives rise to a functor $\mathbb{T}_\Delta \to \mathbb{T}_{\Delta'}$ mapping an object $(\Theta; \vartheta; f)$ to $(\Theta; \vartheta; f\nu)$, and this induces a functor $\nu^* : \mathsf{GR}[\Delta'] \to \mathsf{GR}[\Delta]$ by $(\nu^* F)(\Theta; \vartheta; f) = F(\Theta; \vartheta; f\nu)$. This functor extends to a morphism of CwFs [13], in particular it maps a type A in context Γ in the CwF structure of $\mathsf{GR}[\Delta']$ to a type $\nu^* A$ in context $\nu^* \Gamma$ the CwF structure of $\mathsf{GR}[\Delta]$, and likewise for terms. For example, $(\nu^* A)(\gamma)$ for $\gamma \in \nu^* \Gamma(\Theta; \vartheta; f) = \Gamma(\Theta; \vartheta; f\nu)$ is defined as $A\gamma$. Moreover, this map commutes on the nose with comprehension and substitution. For example, if A is a type in context Γ in $\mathsf{GR}[\Delta']$ and $\gamma : \Gamma' \to \Gamma$, then $(\nu^* A)[\nu^* \gamma] = \nu^*(A[\gamma])$. Moreover, it commutes with $\blacktriangleright$ in the following sense.

▶ **Lemma 7.** *If $\nu : \Delta' \to \Delta$ and $\kappa \in \Delta'$ then $\nu^* \circ \blacktriangleright^\kappa = \blacktriangleright^{\nu(\kappa)} \circ \nu^*$.*

The interpretation of a substitution (ν, σ) is a morphism

$$\llbracket (\nu, \sigma) \rrbracket : \llbracket \Gamma \vdash_\Delta \rrbracket \to \nu^* \llbracket \Gamma \vdash'_{\Delta'} \rrbracket$$

in $\mathsf{GR}[\Delta]$, which we will define below. But first we state the substitution lemma, which must be proved by induction on terms and types simultaneously with the definition of the interpretation, as is standard for models of dependent type theory.

▶ **Lemma 8.** *Let $(\nu, \sigma) : \Gamma \vdash_\Delta \to \Gamma' \vdash_{\Delta'}$ be a substitution and let $\Gamma' \vdash_{\Delta'} J$ be a judgement of a wellformed type or a typing judgement. Then $\llbracket J\,(\nu, \sigma) \rrbracket = (\nu^* \llbracket J \rrbracket)[\llbracket (\nu, \sigma) \rrbracket]$.*

The main difficulty for defining the interpretation of substitutions is that the operator $\blacktriangleleft^\kappa$ does not commute with clock substitutions in the sense that $\nu^*(\blacktriangleleft^\kappa \Gamma)$ is not necessarily equal to $\blacktriangleleft^{\nu(\kappa)}(\nu^*\Gamma)$. However, we can define a map in the relevant direction:

$$e_\Gamma^{\kappa,\nu} = \epsilon_{\nu^*(\blacktriangleleft^\kappa\Gamma)}^{\nu(\kappa)} \circ \blacktriangleleft^{\nu(\kappa)} \nu^*(\eta_\Gamma^\kappa) : \blacktriangleleft^{\nu(\kappa)}(\nu^*\Gamma) \to \nu^*(\blacktriangleleft^\kappa \Gamma)$$

where $\eta_\Gamma^\kappa : \Gamma \to \blacktriangleright^\kappa \blacktriangleleft^\kappa \Gamma$ is the unit of the adjunction and

$$\epsilon_{\nu^*(\blacktriangleleft^\kappa\Gamma)}^{\nu(\kappa)} : \blacktriangleleft^{\nu(\kappa)} \blacktriangleright^{\nu(\kappa)} \nu^*(\blacktriangleleft^\kappa \Gamma) \to \nu^*(\blacktriangleleft^\kappa \Gamma)$$

is the counit. The composition type checks since $\nu^* \blacktriangleright^\kappa = \blacktriangleright^{\nu(\kappa)} \nu^*$.

The map $e^{\kappa,\nu}$ has a simple description in the model: $e_\Gamma^{\kappa,\nu}$ maps an element (X,γ) in

$$\blacktriangleleft^{\nu(\kappa)}(\nu^*\Gamma)(\Theta;\vartheta;f)$$

$$= \coprod_{\substack{\nu(k)\in X \\ f(X)=f(\nu(\kappa))}} (\nu^*\Gamma)(\Theta,\#_\Theta;\vartheta[\#_\Theta \mapsto \vartheta((f\circ\nu)(\kappa))+1];f[X\mapsto\#_\Theta])$$

$$= \coprod_{\substack{\nu(k)\in X \\ f(X)=f(\nu(\kappa))}} \Gamma(\Theta,\#_\Theta;\vartheta[\#_\Theta \mapsto \vartheta((f\circ\nu)(\kappa))+1];f\circ\nu[\nu^{-1}X\mapsto\#_\Theta])$$

to $(\nu^{-1}X,\gamma)$ in the set $\nu^*(\blacktriangleleft^\kappa \Gamma)(\Theta;\vartheta;f)$ which equals

$$\blacktriangleleft^\kappa \Gamma(\Theta;\vartheta;f\circ\nu) = \coprod_{\substack{\kappa\in Y \\ f(\nu(Y))=f(\nu(\kappa))}} \Gamma(\Theta,\#_\Theta;\vartheta[\#_\Theta \mapsto \vartheta((f\circ\nu)(\kappa))+1];f\circ\nu[Y\mapsto\#_\Theta])$$

The interpretation of substitutions is defined as

$$[\![\cdot]\!] = x \mapsto \star$$
$$[\![(\nu,\sigma[x\mapsto t])]\!] = \langle[\![(\nu,\sigma)]\!],[\![t]\!]\rangle$$
$$[\![(\nu,\sigma[\beta\mapsto\alpha])]\!] = e_{[\![\Gamma']\!]}^\kappa \circ \blacktriangleleft^{\nu(\kappa)} [\![(\nu,\sigma)]\!] \circ \mathsf{p}_{\Gamma''}$$
$$[\![(\nu[\kappa\mapsto\nu(\kappa')],\sigma[\alpha\mapsto\diamond])]\!] = \nu^* [\![([\kappa\mapsto\kappa'],[\alpha\mapsto\diamond])]\!] \circ [\![(\nu,\sigma)]\!]$$

where we have assumed the types as in the rules for forming substitutions and $\mathsf{p}_{\Gamma''}$ is the context projection defined as in Section 2.2. The last case uses $[\![([\kappa\mapsto\kappa'],[\alpha\mapsto\diamond])]\!]$ which will be defined in Section 7 below.

The rest of this section is a sketch proof of the substitution lemma for the fragment of CloTT modelled so far, i.e., excluding quantification over clocks and $\diamond$. As we extend the interpretation we will also extend the proof of the substitution lemma.

The proof is by induction over judgements and is simultaneous with the definition of the interpretation of substitutions. The cases of standard dependent type theory (dependent functions and sums, identity types) can be essentially reduced to the standard proof of the substitution lemma for dependent type theory in presheaf models as follows (although the non-standard notion of substitution requires some new lemmas). Since $\mathsf{GR}[\Delta]$ is the object of presheaves over $\mathbb{T}_\Delta$, the category of elements of $[\![\Delta]\!] = \mathsf{Clk}^\Delta$, given a context $\Gamma \vdash_\Delta$ one can form the comprehension $[\![\Delta]\!].[\![\Gamma]\!]$ as an object of GR. Types and terms in context $[\![\Delta]\!].[\![\Gamma]\!]$ in the CwF structure associated to GR are then in bijective correspondence with those over $[\![\Gamma]\!]$ in the CwF structure of $\mathsf{GR}[\Delta]$. A substitution $(\nu,\sigma) : \Gamma \vdash_\Delta \to \Gamma' \vdash_{\Delta'}$ induces a morphism $[\![\Delta]\!].[\![\Gamma]\!] \to [\![\Delta']\!].[\![\Gamma']\!]$ in GR and the substitution defined above corresponds to substitution along this map. Thus the interpretation of the standard type theoretic constructions are

the same as the standard ones in the presheaf model GR, and the corresponding cases of the substitution lemma can be proved similarly to the standard proof of the substitution lemma for presheaf models of type theory.

The cases corresponding to the constructions from the tick calculus follow from the following two equations.

$$\blacktriangleright^{\nu(\kappa)} e^{\kappa,\nu} \circ \eta^{\nu(\kappa)}_{\nu^*} = \nu^*(\eta^\kappa) \qquad\qquad : \nu^* \to \blacktriangleright^{\nu(\kappa)} \nu^* \blacktriangleleft^\kappa = \nu^* \blacktriangleright^\kappa \blacktriangleleft^\kappa \tag{3}$$

$$\nu^*(\epsilon^\kappa) \circ e^{\kappa,\nu}_{\blacktriangleright^\kappa} = \epsilon^{\nu(\kappa)}_{\nu^*} \qquad\qquad : \blacktriangleleft^{\nu(\kappa)} \nu^* \blacktriangleright^\kappa = \blacktriangleleft^{\nu(\kappa)} \blacktriangleright^{\nu(\kappa)} \nu^* \to \nu^* \tag{4}$$

For example, the case of $\triangleright(\alpha : \kappa).A$ can be proved as follows (writing $[\![\sigma]\!]$ for $[\![(\nu,\sigma)]\!]$)

$$\nu^*[\![\Gamma' \vdash_{\Delta'} \triangleright(\alpha:\kappa).A]\!][[\![\sigma]\!]] = \nu^*\left(\left(\blacktriangleright^\kappa [\![\Gamma',\alpha:\kappa\vdash_{\Delta'} A]\!]\right)[\eta^\kappa_{[\![\Gamma']\!]}]\right)[[\![\sigma]\!]]$$

$$= \left(\blacktriangleright^{\nu(\kappa)} \nu^*[\![\Gamma',\alpha:\kappa\vdash_{\Delta'} A]\!]\right)[\nu^*\eta^\kappa_{[\![\Gamma']\!]}][[\![\sigma]\!]]$$

$$= \left(\blacktriangleright^{\nu(\kappa)} \nu^*[\![\Gamma',\alpha:\kappa\vdash_{\Delta'} A]\!]\right)[\blacktriangleright^{\nu(\kappa)} e^\kappa \circ \eta^{\nu(\kappa)}_{\nu^*[\![\Gamma']\!]} \circ [\![\sigma]\!]]$$

$$= \left(\blacktriangleright^{\nu(\kappa)} \nu^*[\![\Gamma',\alpha:\kappa\vdash_{\Delta'} A]\!]\right)[\blacktriangleright^{\nu(\kappa)} (e^\kappa \circ \blacktriangleleft^{\nu(\kappa)} [\![\sigma]\!]) \circ \eta^{\nu(\kappa)}_{\nu^*[\![\Gamma']\!]}]$$

$$= \left(\blacktriangleright^{\nu(\kappa)} (\nu^*[\![\Gamma',\alpha:\kappa\vdash_{\Delta'} A]\!])[e^\kappa \circ \blacktriangleleft^{\nu(\kappa)} [\![\sigma]\!]]\right)[\eta^{\nu(\kappa)}_{\nu^*[\![\Gamma']\!]}]$$

$$= \left(\blacktriangleright^{\nu(\kappa)} [\![\Gamma,\beta:\nu(\kappa)\vdash_\Delta A(\nu,\sigma[\alpha\mapsto\beta])]\!]\right)[\eta^{\nu(\kappa)}_{\nu^*[\![\Gamma']\!]}]$$

$$= [\![\Gamma \vdash_\Delta \triangleright(\beta:\nu(\kappa)).(A(\nu,\sigma[\alpha\mapsto\beta]))]\!]$$

$$= [\![\Gamma \vdash_\Delta (\triangleright(\alpha:\kappa).A)(\nu,\sigma)]\!]$$

6 Interpretation of clock quantification

Universal quantification over clocks should be modelled as a right adjoint to the semantic correspondent to clock weakening. Syntactically, clock weakening from context $\Gamma \vdash_\Delta$ to $\Gamma \vdash_{\Delta,\kappa}$ corresponds to the substitution (i, id_Γ), where i is the inclusion of Δ into Δ, κ. Recall from Section 5 that types and terms in context $[\![\Gamma \vdash_\Delta]\!]$ in the CwF structure of $\mathsf{GR}[\Delta]$ correspond to types and terms in context $[\![\Delta]\!].[\![\Gamma \vdash_\Delta]\!]$ in GR, and recall that $[\![\Delta]\!] = \mathrm{Clk}^\Delta$. By the substitution lemma, clock weakening from context $\Gamma \vdash_\Delta$ to $\Gamma \vdash_{\Delta,\kappa}$ corresponds to substitution along the composition

$$[\![\Delta,\kappa]\!].[\![\Gamma \vdash_{\Delta,\kappa}]\!] \xrightarrow{[\![\Delta,\kappa]\!].[\![(i,\mathsf{id}_\Gamma)]\!]} [\![\Delta,\kappa]\!].i^*[\![\Gamma \vdash_\Delta]\!] \to [\![\Delta]\!].[\![\Gamma \vdash_\Delta]\!]$$

All such substitutions have right adjoints, but to get a simple description of this (and to satisfy the substitution lemma), we will give a concrete description which can be briefly described as follows: To model $\forall \kappa.A$, open a fresh semantic clock #, map κ to # and take the limit of $[\![A]\!]$ as # ranges over all natural numbers. To type this description we need the following lemma.

▶ **Lemma 9.** *Let $(\Theta; \vartheta; f)$ be an object of $\mathsf{GR}[\Delta]$, let # be fresh for Θ and let $\iota : \Theta \to \Theta, \#$ be the inclusion. The component of $[\![(i, \mathsf{id}_\Gamma)]\!]$ at $(\Theta, \#; \vartheta[\# \mapsto n]; f[\kappa \mapsto \#])$ is an isomorphism*

$$[\![\Gamma \vdash_{\Delta,\kappa}]\!]_{(\Theta,\#;\vartheta[\#\mapsto n];f[\kappa\mapsto\#])} \to i^*[\![\Gamma \vdash_\Delta]\!]_{(\Theta,\#;\vartheta[\#\mapsto n];f[\kappa\mapsto\#])} = [\![\Gamma \vdash_\Delta]\!]_{(\Theta,\#;\vartheta[\#\mapsto n];\iota f)}$$

If $\kappa' \in \Delta$, the inverse to $[\![(i, \mathsf{id}_\Gamma)]\!]_{(\Theta,\#;\vartheta[\#\mapsto n];f[\kappa\mapsto\#])}$ is given by the component of

$$i^*([\![(\mathsf{id}_\Delta[\kappa\mapsto\kappa'],\mathsf{id}_\Gamma)]\!]) : i^*[\![\Gamma \vdash_\Delta]\!] \to i^*(\mathsf{id}_\Delta[\kappa\mapsto\kappa'])^*[\![\Gamma \vdash_{\Delta,\kappa}]\!] = [\![\Gamma \vdash_{\Delta,\kappa}]\!]$$

at $(\Theta, \#; \vartheta[\# \mapsto n]; f[\kappa \mapsto \#])$).

We can now define the interpretation of universal quantification over clocks:

$$\llbracket \Gamma \vdash_\Delta \forall \kappa.A \text{ type} \rrbracket_{(\Theta;\vartheta;f)}(\gamma) \overset{\text{def}}{=}$$

$$\{(\omega_n) \in \prod_{n \in \mathbb{N}} \llbracket \Gamma \vdash_{\Delta,\kappa} A \text{ type} \rrbracket_{(\Theta,\#;\vartheta[\#\mapsto n];f[\kappa \mapsto \#])}(\llbracket (i,\text{id}_\Gamma) \rrbracket^{-1}(\iota \cdot \gamma)) \mid \forall n.\omega_n = (\omega_{n+1})|_n\}$$

Here, by assumption $\gamma \in \llbracket \Gamma \vdash_\Delta \rrbracket_{(\Theta;\vartheta;f)}(\gamma)$ and so

$$\iota \cdot \gamma \in \llbracket \Gamma \vdash_\Delta \rrbracket_{(\Theta,\#;\vartheta[\#\mapsto n];\iota f)} = i^* \llbracket \Gamma \vdash_\Delta \rrbracket_{(\Theta,\#;\vartheta[\#\mapsto n];f[\kappa \mapsto \#])}$$

which means that $\llbracket (i,\text{id}_\Gamma) \rrbracket^{-1}(\iota \cdot \gamma) \in \llbracket \Gamma \vdash_{\Delta,\kappa} \rrbracket_{(\Theta,\#;\vartheta[\#\mapsto n];f[\kappa \mapsto \#])}$ and thus the type of each ω_n is a welldefined set. In the condition for the families, $(\omega_{n+1})|_n$ is the restriction of ω_{n+1} along the map $(\Theta,\#;\vartheta[\# \mapsto n+1];f[\kappa \mapsto \#]) \to (\Theta,\#;\vartheta[\# \mapsto n];f[\kappa \mapsto \#])$ given by the identity on $\Theta,\#$. For the interpretation of terms define

$$\llbracket \Gamma \vdash_\Delta \Lambda\kappa.t : \forall \kappa.A \rrbracket_{(\Theta;\vartheta;f)} = (\llbracket \Gamma \vdash_{\Delta,\kappa} t : A \rrbracket_{(\Theta,\#;\vartheta[\#\mapsto n];\iota f)}(\llbracket (i,\text{id}_\Gamma) \rrbracket^{-1}(\iota \cdot \gamma)))_n$$

$$\llbracket \Gamma \vdash_\Delta t[\kappa'] : A[\kappa'/\kappa] \rrbracket_{(\Theta;\vartheta;f)} = [\# \mapsto f(\kappa')] \cdot (\llbracket \Gamma \vdash_\Delta t : \forall \kappa.A \rrbracket_{(\Theta;\vartheta;f)}(\gamma))_{\vartheta(f(\kappa'))}$$

To see that the latter type checks, note first that

$$\llbracket \Gamma \vdash_\Delta t : \forall \kappa.A \rrbracket_{(\Theta;\vartheta;f)}(\gamma))_{\vartheta(f(\kappa'))} \in \llbracket \Gamma \vdash_{\Delta,\kappa} A \text{ type} \rrbracket_{(\Theta,\#;\vartheta[\#\mapsto \vartheta(f(\kappa'))];f[\kappa \mapsto \#])}(\llbracket (i,\text{id}_\Gamma) \rrbracket^{-1}(\iota \cdot \gamma))$$

and since $[\# \mapsto f(\kappa')] : (\Theta,\#;\vartheta[\# \mapsto \vartheta(f(\kappa'))];f[\kappa \mapsto \#]) \to (\Theta;\vartheta;f[\kappa \mapsto f(\kappa')])$, the right hand side of the definition is in

$$\llbracket \Gamma \vdash_{\Delta,\kappa} A \text{ type} \rrbracket_{(\Theta;\vartheta;f[\kappa\mapsto f(\kappa')])}([\# \mapsto f(\kappa')] \cdot \llbracket (i,\text{id}_\Gamma) \rrbracket^{-1}(\iota \cdot \gamma))$$

$$=(\text{id}_\Delta[\kappa \mapsto \kappa'])^*(\llbracket \Gamma \vdash_{\Delta,\kappa} A \text{ type} \rrbracket)_{(\Theta;\vartheta;f)}([\# \mapsto f(\kappa')] \cdot \llbracket (i,\text{id}_\Gamma) \rrbracket^{-1}(\iota \cdot \gamma))$$

$$=(\text{id}_\Delta[\kappa \mapsto \kappa'])^*(\llbracket \Gamma \vdash_{\Delta,\kappa} A \text{ type} \rrbracket)_{(\Theta;\vartheta;f)}([\# \mapsto f(\kappa')] \cdot \llbracket (\text{id}_\Delta[\kappa \mapsto \kappa'],\text{id}_\Gamma) \rrbracket(\iota \cdot \gamma))$$

$$=(\text{id}_\Delta[\kappa \mapsto \kappa'])^*(\llbracket \Gamma \vdash_{\Delta,\kappa} A \text{ type} \rrbracket)_{(\Theta;\vartheta;f)}(\llbracket (\text{id}_\Delta[\kappa \mapsto \kappa'],\text{id}_\Gamma) \rrbracket([\# \mapsto f(\kappa')] \cdot \iota \cdot \gamma))$$

$$=(\text{id}_\Delta[\kappa \mapsto \kappa'])^*(\llbracket \Gamma \vdash_{\Delta,\kappa} A \text{ type} \rrbracket)_{(\Theta;\vartheta;f)}(\llbracket (\text{id}_\Delta[\kappa \mapsto \kappa'],\text{id}_\Gamma) \rrbracket(\gamma))$$

$$=\llbracket \Gamma \vdash_\Delta A[\kappa'/\kappa] \text{ type} \rrbracket_{(\Theta;\vartheta;f)}(\gamma)$$

where the last equality is by the substitution lemma.

▶ **Lemma 10.** *The β and η rules for universal quantification over clocks are sound for the interpretation.*

7 Interpretation of ◇

To interpret the rule for the tick constant ◇, we define a substitution

$$\llbracket ([\kappa \mapsto \kappa'], [\alpha \mapsto \diamond]) \rrbracket : \llbracket \Gamma \vdash_\Delta \rrbracket \to [\kappa \mapsto \kappa']^* \llbracket \Gamma, \alpha : \kappa \vdash_{\Delta,\kappa} \rrbracket$$

for every context $\Gamma \vdash_\Delta$ with $\kappa \notin \Delta$. The interpretation of application to ◇ can then be defined as

$$\llbracket \Gamma \vdash_\Delta t[\kappa'/\kappa][\diamond] : A[\kappa'/\kappa][\diamond/\alpha] \rrbracket = ([\kappa \mapsto \kappa']^* \llbracket \Gamma, \alpha : \kappa \vdash_{\Delta,\kappa} t : A \rrbracket)[\llbracket ([\kappa \mapsto \kappa'], [\alpha \mapsto \diamond]) \rrbracket]$$

which has type

$$([\kappa \mapsto \kappa']^* \llbracket \Gamma, \alpha : \kappa \vdash_{\Delta,\kappa} A \text{ type} \rrbracket)[\llbracket ([\kappa \mapsto \kappa'], [\alpha \mapsto \diamond]) \rrbracket]$$

which equals the required $[\![\Gamma, \alpha : \kappa \vdash_{\Delta,\kappa} A[\kappa'/\kappa][\diamond/\alpha] \text{ type}]\!]$ by the substitution lemma.

Suppose $\gamma \in [\![\Gamma \vdash_\Delta]\!]_{(\Theta;\vartheta;f)}$. We define

$$[\![([\kappa \mapsto \kappa'], [\alpha \mapsto \diamond])]\!](\gamma) \stackrel{\text{def}}{=} \left(\{\kappa\}, [\![(i, \mathsf{id}_\Gamma)]\!]^{-1}(\iota \cdot \gamma)\right)$$

where $\iota : (\Theta; \vartheta; f) \to (\Theta, \#; \vartheta[\# \mapsto n+1]; f)$ is the inclusion for $n = \vartheta(f(\kappa'))$. We must show that this defines an element of $[\kappa \mapsto \kappa']^*[\![\Gamma, \alpha : \kappa \vdash_{\Delta,\kappa}]\!]_{(\Theta;\vartheta;f)}$. To see this, note first that

$$\iota \cdot \gamma \in [\![\Gamma \vdash_\Delta]\!]_{(\Theta,\#;\vartheta[\# \mapsto n+1];f)} = i^*[\![\Gamma \vdash_\Delta]\!]_{(\Theta,\#;\vartheta[\# \mapsto n+1];f[\kappa \mapsto \#])}$$

so by Lemma 9,

$$[\![(i, \mathsf{id}_\Gamma)]\!]^{-1}(\iota \cdot \gamma) \in [\![\Gamma \vdash_{\Delta,\kappa}]\!]_{(\Theta,\#;\vartheta[\# \mapsto n+1];f[\kappa \mapsto \#])}$$

and

$$\left(\{\kappa\}, [\![(i, \mathsf{id}_\Gamma)]\!]^{-1}(\iota \cdot \gamma)\right) \in \coprod_{\kappa \in X \subset f^{-1}(f(\kappa))} [\![\Gamma \vdash_{\Delta,\kappa}]\!]_{(\Theta,\#;\vartheta[\# \mapsto n+1];f[\kappa \mapsto f(\kappa')][X \mapsto \#])}$$

$$= [\![\Gamma, \alpha : \kappa \vdash_{\Delta,\kappa}]\!]_{(\Theta;\vartheta;f[\kappa \mapsto f(\kappa')])}$$

$$= [\kappa \mapsto \kappa']^*[\![\Gamma, \alpha : \kappa \vdash_{\Delta,\kappa}]\!]_{(\Theta;\vartheta;f)}$$

We note that this satisfies the equality for $\diamond$ in Figure 1.

▶ **Lemma 11.** *The interpretations of* $(\mathsf{dfix}^{\kappa'} t)[\diamond]$ *and* $t(\mathsf{dfix}^{\kappa'} t)$ *are equal.*

7.1 Welldefinedness

As mentioned in Section 3 the unusual typing rule for $t[\diamond]$ introduces a problem of welldefinedness of the interpretation: If t is a term, a proof of the typing judgement $\Gamma \vdash_\Delta t[\diamond] : A[\diamond/\alpha]$ consists of a term s such that $s[\kappa'/\kappa] = t$, a type B such that $B[\kappa'/\kappa] = A$ and a proof of a typing judgement $\Gamma \vdash_{\Delta,\kappa} s : B$. In general there may be different possible choices of s and B, but the next lemma states that the interpretation of the term $t[\diamond]$ is independent of this choice. This means that the interpretation of a welltyped term is a welldefined object, independent of the choice of typing derivation.

▶ **Proposition 12.** *If* $\Gamma \vdash_{\Delta,\kappa} s : B$ *and* $\Gamma \vdash_{\Delta,\kappa} u : C$ *are such that* $\Gamma \vdash_\Delta s[\kappa'/\kappa] : B[\kappa'/\kappa]$ *is equal to* $\Gamma \vdash_\Delta u[\kappa'/\kappa] : C[\kappa'/\kappa]$ *then*

$$[\kappa \mapsto \kappa']^*[\![s[\alpha]]\!][\![([\kappa \mapsto \kappa'], [\alpha \mapsto \diamond])]\!] = [\kappa \mapsto \kappa']^*[\![u[\alpha]]\!][\![([\kappa \mapsto \kappa'], [\alpha \mapsto \diamond])]\!]$$

Proof. The assumption implies that also

$$\Gamma, \alpha : \kappa' \vdash_\Delta (s[\alpha])[\kappa'/\kappa] : B[\kappa'/\kappa] \qquad \text{and} \qquad \Gamma, \alpha : \kappa' \vdash_\Delta (u[\alpha])[\kappa'/\kappa] : B[\kappa'/\kappa]$$

are equal, and so by the substitution lemma

$$([\kappa \mapsto \kappa'][\![s[\alpha]]\!])[\![(i, \mathsf{id}_\Gamma[\alpha \mapsto \alpha])]\!] = ([\kappa \mapsto \kappa'][\![u[\alpha]]\!])[\![(i, \mathsf{id}_\Gamma[\alpha \mapsto \alpha])]\!]$$

Now, if $\gamma \in \llbracket \Gamma \vdash_\Delta \rrbracket_{(\Theta;\vartheta;f)}$ then

$$[\kappa \mapsto \kappa']^* \llbracket s\,[\alpha] \rrbracket \llbracket \llbracket ([\kappa \mapsto \kappa'], [\alpha \mapsto \diamond]) \rrbracket \rrbracket (\gamma)$$

$$= [\kappa \mapsto \kappa']^* \llbracket s\,[\alpha] \rrbracket \left(\{\kappa\}, \llbracket (i, \mathsf{id}_\Gamma) \rrbracket^{-1}(\iota \cdot \gamma) \right)$$

$$= [\kappa \mapsto \kappa']^* \llbracket s\,[\alpha] \rrbracket \left(\{\kappa\}, \llbracket ([\kappa \mapsto \kappa'], \mathsf{id}_\Gamma) \rrbracket (\iota \cdot \gamma) \right)$$

$$= [\kappa \mapsto \kappa']^* \llbracket s\,[\alpha] \rrbracket \llbracket ([\kappa \mapsto \kappa'], \mathsf{id}_\Gamma[\alpha \mapsto \alpha]) \rrbracket (\{\kappa\}, (\iota \cdot \gamma))$$

$$= [\kappa \mapsto \kappa']^* \llbracket u\,[\alpha] \rrbracket \llbracket ([\kappa \mapsto \kappa'], \mathsf{id}_\Gamma[\alpha \mapsto \alpha]) \rrbracket (\{\kappa\}, (\iota \cdot \gamma))$$

$$= [\kappa \mapsto \kappa']^* \llbracket u\,[\alpha] \rrbracket \llbracket \llbracket ([\kappa \mapsto \kappa'], [\alpha \mapsto \diamond]) \rrbracket \rrbracket (\gamma) \qquad \blacktriangleleft$$

8 Clock irrelevance

We now show how to model the clock irrelevance axiom (1). For this it suffices to show that if $\Gamma \vdash_\Delta t : \forall \kappa.A$, $\Gamma \vdash_\Delta A$ type and $\kappa', \kappa'' \in \Delta$ then

$$\llbracket \Gamma \vdash_\Delta t[\kappa'] : A \rrbracket = \llbracket \Gamma \vdash_\Delta t[\kappa''] : A \rrbracket$$

Recall that for $\gamma \in \llbracket \Gamma \vdash_\Delta \rrbracket_{(\Theta;\vartheta;f)}$

$$\llbracket \Gamma \vdash_\Delta t[\kappa'] : A \rrbracket_{(\Theta;\vartheta;f)}(\gamma) = [\# \mapsto f(\kappa')] \cdot (\llbracket \Gamma \vdash_\Delta t : \forall \kappa.A \rrbracket_{(\Theta;\vartheta;f)}(\gamma))_n$$

where $n = \vartheta(f(\kappa'))$. Here the element $(\llbracket \Gamma \vdash_\Delta t : \forall \kappa.A \rrbracket_{(\Theta;\vartheta;f)}(\gamma))_n$ lives in

$$\llbracket \Gamma \vdash_{\Delta,\kappa} A\ \text{type} \rrbracket_{(\Theta,\#;\vartheta[\# \mapsto n];f[\kappa \mapsto \#])}(\llbracket (i, \mathsf{id}_\Gamma) \rrbracket^{-1}(\iota \cdot \gamma))$$

$$= i^*(\llbracket \Gamma \vdash_\Delta A\ \text{type} \rrbracket)_{(\Theta,\#;\vartheta[\# \mapsto n];f[\kappa \mapsto \#])}(\llbracket (i, \mathsf{id}_\Gamma) \rrbracket(\llbracket (i, \mathsf{id}_\Gamma) \rrbracket^{-1}(\iota \cdot \gamma)))$$

$$= \llbracket \Gamma \vdash_\Delta A\ \text{type} \rrbracket_{(\Theta,\#;\vartheta[\# \mapsto n];f)}(\iota \cdot \gamma)$$

Clock irrelevance will follow from the following lemma.

▶ **Lemma 13.** *Suppose $\Gamma \vdash_\Delta A$ type and $\gamma \in \llbracket \Gamma \vdash_\Delta \rrbracket_{(\Theta;\vartheta;f)}$. The map*

$$\iota \cdot (-) : \llbracket \Gamma \vdash_\Delta A\ \text{type} \rrbracket_{(\Theta;\vartheta;f)} \to \llbracket \Gamma \vdash_\Delta A\ \text{type} \rrbracket_{(\Theta,\#;\vartheta[\# \mapsto n];f)}(\iota \cdot \gamma)$$

is an isomorphism.

In particular, there is an element x such that $\iota \cdot x = (\llbracket \Gamma \vdash_\Delta t : \forall \kappa.A \rrbracket_{(\Theta;\vartheta;f)}(\gamma))_n$ and so

$$\llbracket \Gamma \vdash_\Delta t[\kappa'] : A \rrbracket_{(\Theta;\vartheta;f)}(\gamma) = [\# \mapsto f(\kappa')] \cdot \iota \cdot x = x$$

Likewise $\llbracket \Gamma \vdash_\Delta t[\kappa''] : A \rrbracket_{(\Theta;\vartheta;f)}(\gamma) = x$ proving clock irrelevance.

Lemma 13 can be proved by induction on A using the techniques of [10]. In particular, [10] proves that the statement of the lemma is equivalent to the statement that the context projection map $\llbracket \Gamma, x : A \vdash_\Delta \rrbracket \to \llbracket \Gamma \vdash_\Delta \rrbracket$ (considered as a morphism in GR) is *orthogonal* to all objects of the form $y(\lambda, n)$ where y is the yoneda embedding. Here, orthogonality means that for all squares as below, there exists a unique filling diagonal:

where π is the second projection. In particular, this implies that the condition is closed under Π- and Σ-types, and substitutions, see [10] for details. This proof can be easily extended to the cases of $\triangleright (\alpha : \kappa).A$ and $\forall \kappa.A$.

9 Conclusion and future work

We have constructed a model of CloTT modelling ticks on clocks using an adjunction where the right adjoint extends to types and terms. The description of the left adjoint $\blacktriangleleft^\kappa$ is fairly heavy to work with, but by abstracting away the required properties needed to model the tick calculus, the model can be described in reasonable space.

Future work includes extending the model to universes, which we expect to be easy using the universes constructed in [10]. As noted there the axiom of clock irrelevance forces universes to be indexed by syntactic clock contexts. Fortunately, we can model a notion of universe polymorphism in the clock context: inclusions of clock contexts induce inclusion of universes, and these commute with type constructions on the nose. These results, however, must be adapted to CloTT, in particular the code for the $\triangleright$ type constructor should be extended to the tick abstracting generalisation used in CloTT. We expect this to be a simple adaptation. Using universes, guarded recursive types can be encoded [6], indeed these can also be added as primitive, given that many of them exist in the model [10].

Our motivation for constructing this model is to study extensions of CloTT. In particular, we would like to extend CloTT with path types as in [5]. This requires an adaptation of the model to the cubical setting, using $\mathbb{T}$ indexed families of cubical sets [11] rather than just sets.

References

1 A. W. Appel and D. A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst*, 23(5):657–683, 2001.

2 R. Atkey and C. McBride. Productive coprogramming with guarded recursion. In *ICFP*, pages 197–208. ACM, 2013.

3 P. Bahr, H. B. Grathwohl, and R. E. Møgelberg. The clocks are ticking: No more delays! In *LICS*, pages 1–12. IEEE, 2017.

4 J.-P. Bernardy, T. Coquand, and G. Moulin. A presheaf model of parametric type theory. *Electronic Notes in Theoretical Computer Science*, 319:67–82, 2015.

5 L. Birkedal, A. Bizjak, R. Clouston, H. B. Grathwohl, B. Spitters, and A. Vezzosi. Guarded cubical type theory: Path equality for guarded recursion. In *CSL*, 2016.

6 L. Birkedal and R. E. Møgelberg. Intensional type theory with guarded recursive types qua fixed points on universes. In *LICS*, pages 213–222. IEEE, 2013.

7 L. Birkedal, R.E. Møgelberg, J. Schwinghammer, and K. Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *LMCS*, 8(4), 2012.

8 A. Bizjak, L. Birkedal, and M. Miculan. A model of countable nondeterminism in guarded type theory. In *RTA-TLCA*, pages 108–123, 2014.

9 A. Bizjak, H. B. Grathwohl, R. Clouston, R. E. Møgelberg, and L. Birkedal. Guarded dependent type theory with coinductive types. In *FOSSACS*, 2016.

10 A. Bizjak and R. Møgelberg. Denotational semantics for guarded dependent type theory. *arXiv*, 2017. `arXiv:1802.03744`.

11 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:34, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.TYPES.2015.5`.

12 T. Coquand. Infinite objects in type theory. In *International Workshop on Types for Proofs and Programs*, pages 62–78. Springer, 1993.

13 P. Dybjer. Internal type theory. In *International Workshop on Types for Proofs and Programs*, pages 120–134. Springer, 1995.

14 P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

15 C. McBride and R. Paterson. Applicative programming with effects. *Journal of functional programming*, 18(1):1–13, 2008.

16 H. Nakano. A modality for recursion. In *LICS*, pages 255–266, 2000.

17 A. M. Pitts, J. Matthiesen, and J. Derikx. A dependent type theory with abstractable names. *Electronic Notes in Theoretical Computer Science*, 312:19–50, 2015.

18 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

Call-by-Name Gradual Type Theory

Max S. New[1]

Northeastern University, Boston, USA
maxnew@ccs.neu.edu

Daniel R. Licata[2]

Wesleyan University, Middletown, USA
dlicata@wesleyan.edu

─── **Abstract** ───────────────────

We present *gradual type theory*, a logic and type theory for call-by-name gradual typing. We define the central constructions of gradual typing (the dynamic type, type casts and type error) in a novel way, by universal properties relative to new judgments for *gradual type and term dynamism*. These dynamism judgements build on prior work in blame calculi and on the "gradual guarantee" theorem of gradual typing. Combined with the ordinary extensionality (η) principles that type theory provides, we show that most of the standard operational behavior of casts is *uniquely determined* by the gradual guarantee. This provides a semantic justification for the definitions of casts, and shows that non-standard definitions of casts must violate these principles. Our type theory is the internal language of a certain class of preorder categories called *equipments*. We give a general construction of an equipment interpreting gradual type theory from a 2-category representing non-gradual types and programs, which is a semantic analogue of the interpretation of gradual typing using contracts, and use it to build some concrete domain-theoretic models of gradual typing.

─────────────────────────

1 Introduction

Gradually typed languages allow for static and dynamic programming styles within the same language. They are designed with twin goals of allowing easy interoperability between static and dynamic portions of a codebase and facilitating a smooth transition from dynamic to static typing. This allows for the introduction of new typing features to legacy languages and codebases without the enormous manual effort currently necessary to migrate code from a dynamically typed language to a fully statically typed language. Gradual typing allows exploratory programming and prototyping to be done in a forgiving, dynamically typed style, while later that code can be typed to ease readability and refactoring. Due to this appeal, there has been a great deal of research on extending gradual typing [29, 25] to numerous language features such as parametric polymorphism [1, 12], effect tracking [2], typestate [33], session types [11], and refinement types [14]. Almost all work on gradual typing is based solely on operational semantics, and recent work such as [24] has codified some of the central design principles of gradual typing in an operational setting. In this paper, we are interested in complementing this operational work with a type-theoretic and category-theoretic analysis of these design principles. We believe this will improve our understanding of gradually typed languages, particularly with respect to principles for reasoning about program equivalence, and assist in designing and evaluating new gradually typed languages.

One of the central design principles for gradual typing is *gradual type soundness*. At its most general, this should mean that the types of the gradually typed language provide the same type-based reasoning that one could reasonably expect from a similar statically typed language, i.e. one with effects. While this has previously been defined using operational semantics and a notion of *blame* [31], the idea of soundness we consider here is that the types should provide the same extensionality (η) principles as in a statically typed language. This way, programmers can reason about the "typed" parts of gradual programs in the same way as in a fully static language. This definition fits nicely with a category-theoretic perspective, because the β and η principles correspond to definitions of connectives by a *universal property*. The second design principle is the *gradual guarantee* [24], which we will refer to as *graduality* (by analogy with parametricity). Informally, graduality of a language means that syntactic changes from dynamic to static typing (or vice-versa) should result in simple, predictable changes to the semantics of a term. More specifically, if a portion of a program is made "more static"/"less dynamic" then the new program should either have the same behavior or result in a runtime type error. Other observable behavior such as values produced, I/O actions performed or termination should not be changed.

In this paper, we codify these two principles of soundness and graduality *directly* into a logical syntax we dub (call-by-name) *Gradual Type Theory* (Section 2). For graduality, we develop a logic of *type and term dynamism* that can be used to reason about the relationship between "more dynamic" and "less dynamic" versions of a program, and to give novel specifications/universal properties for the dynamic type, type errors, and runtime type casts of a gradually typed language. These universal properties extend the judgmental approach to type theory (see [16, 21]) to the key features of gradual typing. For soundness, we assert β and η principles as axioms of term dynamism, so it can also be used to reason about programs' behavior. Furthermore, using the η principles for types, we show that most of the operational rules of runtime casts of existing (call-by-name) gradually typed languages are *uniquely determined* by these constraints of soundness and graduality (Section 3). For example, uniqueness implies that any enforcement scheme in a specific gradually typed language that is *not* equivalent to the standard "wrapping" ones *must* violate either soundness or graduality.

We have chosen call-by-name because it is a simple setting with the necessary η principles (for negative types) to illustrate our technique; we leave a call-by-value extension to future work.

We give a sound and complete category theoretic semantics for gradual type theory in terms of certain *preorder categories* (double categories where one direction is thin) (Section 4). We show that the contract interpretation of gradual typing [30] can be understood as a tool for constructing models (Section 5): starting from some existing language/category C, we first implement casts as suitable pairs of functions/morphisms from C, and then equip every type with canonical casts to the dynamic type. We apply this to construct some concrete models in domains (Section 6). Conceptually, gradual type theory is analogous to Moggi's *monadic metalanguage* [18]: it clarifies general principles present in many different programming languages; it is the internal language of a quite general class of category-theoretic structures; and, for a specific language, a number of useful results can be proved all at once by showing that a logical relation over it is a model of the type theory.

A logic of dynamism and casts.   Before proceeding to the technical details, we explain at a high level how our type theory accounts for two key features of gradual typing: graduality and casts. The "gradual guarantee" as defined in [24] applies to a surface language where runtime type casts are implicitly inserted based on type annotations, but we will focus here on an analysis of fully elaborated languages, where explicit casts have already been inserted (so our work does not yet address gradual type checking). The gradual guarantee as defined in [24] makes use of a *syntactically less dynamic* ordering on types: the dynamic type (universal domain) ? is the most dynamic, and A is less dynamic than B if B has the same structure as A but some sub-terms are replaced with ? (for example, $A \to (B \times C)$ is less dynamic than $? \to (B \times ?)$, $? \to ?$ and ?). Intuitively, a less dynamic type constrains the behavior of the program more, but consequently gives stronger reasoning principles. This notion is extended to closed well-typed *terms* $t : A$ and $t' : A'$ with A less dynamic than A': t is *syntactically less dynamic* than t' if t is obtained from t' by replacing the input and output type of each type cast with a less (or equally) dynamic type (in [24] this was called "precision"). For example, if $add1 : ? \to \mathbb{N}$ and true : ?, then $add1\,((? \Leftarrow \mathbb{N})(\mathbb{N} \Leftarrow ?)\mathsf{true})$ (cast true from dynamic to $\mathbb{N}$ and back, to assert it is a number) is syntactically less dynamic than $add1\,((? \Leftarrow ?)(? \Leftarrow ?)\mathsf{true})$ (where both casts are the identity). Then the gradual guarantee [24] says that if t is syntactically less dynamic than t', then t is *semantically less dynamic* than t': either t evaluates to a type error (in which case t' may do anything) or t, t' have the same first-order behavior (both diverge or both terminate with t producing a less dynamic value). In the above example, the less dynamic term always errors (because true fails the runtime $\mathbb{N}$ check), while the more dynamic term only errors if $add1$ uses its argument as a number. In contrast, a program that returns a different value than $add1(\mathsf{true})$ does will not be semantically less dynamic than it.

The approach we take in this paper is to give a *syntactic logic* for the *semantic* notion of one term being less dynamic than another, with $\mho$ (type error) the least element, and all term constructors monotone. We call this the *term dynamism relation* $t \sqsubseteq t'$, and it includes not only syntactic changes in type casts, as above, but also equational laws like identity and composition for casts, and $\beta\eta$ rules – so $t \sqsubseteq t'$ intuitively means that t type-errors more than (or as much as) t', but is otherwise equal according to these equational laws. A programming language that is a model of our type theory will therefore be equipped with a semantic $t[\![\sqsubseteq]\!]t'$ relation validating these rules, so $t[\![\sqsubseteq]\!]t'$ if t type-errors more than t' up to these equational and monotonicity laws. In particular, making type cast annotations less dynamic will result

in related programs, and if $[\![\sqsubseteq]\!]$ is adequate (doesn't equate operationally distinguishable terms), then this implies the gradual guarantee [24]. Therefore, we say a model "satisfies graduality" in the same sense that a language satisfies parametricity.

Next, we discuss the relationship between term dynamism and casts/contracts, one of the most novel parts of our theory. Explicit casts in a gradually typed language are typically presented by the syntactic form $(B \Leftarrow A)t$, and their semantics is either defined by various operational reductions that inspect the structure of A and B, or by "contract" translations, which compile a language with casts to another language, where the casts are implemented as ordinary functions. In both cases, the behavior of casts is defined by inspection on types and part of the language definition, with little justification beyond intuition and precedent.

In gradual type theory, on the other hand, the behavior of casts is *not* defined by inspection of types. Rather, we use the new type and term dynamism judgments, which are defined *prior to* casts, to give a few simple and uniform rules specifying casts in all types via a universal property (optimal implementation of a specification). Our methodology requires isolating two special subclasses of casts, upcasts and downcasts. An upcast goes from a "more static" to a "more dynamic" type – for instance $(? \Leftarrow (A \to B))$ is an upcast from a function type up to the dynamic type – whereas a downcast is the opposite, casting to the more static type. We represent the relationship "A is less dynamic than B" by a *type dynamism* judgment $A \sqsubseteq B$ (which corresponds to the "naïve subtyping" of [31]). In gradual type theory, the upcast $\langle B \hookleftarrow A \rangle$ from A to B and the downcast $\langle A \leftarrow B \rangle$ from B to A can be formed whenever $A \sqsubseteq B$. This leaves out certain casts like $(? \times \mathbb{N}) \Leftarrow (\mathbb{N} \times ?)$ where neither type is more dynamic than the other. However, as first recognized in [10], these casts are macro-expressible [6] as a composite of an upcast to the dynamic type and then a downcast from it (define $(B \Leftarrow A)t$ as the composite $\langle B \leftarrow ? \rangle \langle ? \hookleftarrow A \rangle t$).

A key insight is that we can give upcasts and downcasts dual specifications using term dynamism, which say how the casts relate programs to type dynamism. If $A \sqsubseteq B$, then for any term $t : A$, the upcast $\langle B \hookleftarrow A \rangle t : B$ is the *least* dynamic term of type B that is more dynamic than t. In order-theoretic terms, $\langle B \hookleftarrow A \rangle t : B$ is the $\sqsubseteq$-meet of all terms $u : B$ with $t \sqsubseteq u$. Downcasts have a dual interpretation as a $\sqsubseteq$-join. Intuitively, this property means upcast $\langle B \hookleftarrow A \rangle t$ behaves as much as possible like t itself, while supporting the additional interface provided by expanding the type from A to B.

This simple definition has powerful consequences that we explore in Section 3, because it characterizes the upcasts and downcasts up to program equivalence. We show that standard implementations of casts are the *unique* implementations that satisfy β, η and basic congruence rules. In fact, almost all of the standard operational rules of a simple call-by-name gradually typed language are term-dynamism equivalences in gradual type theory. The exception is rules that rely on disjointness of different type connectives (such as $\langle ? \to ? \leftarrow ? \rangle \langle ? \hookleftarrow ? \times ? \rangle t \mapsto \mho$), which are independent, and can be added as axioms.

2   Gradual Type Theory

In this section, we present the rules of gradual type theory (GTT). Gradual type theory presents the types, connectives and casts of gradual typing in a modular, type-theoretic way: the dynamic type and casts are defined by rules using the *judgmental structure* of the type theory, which extends the usual judgmental structure of call-by-name typed lambda calculus with a syntax for type and term dynamism. Since the judgmental structure is as important as these types, we present a bare *preorder type theory* (PTT) with no types first. Then we can modularly define what it means for this theory to have a dynamic type, casts, functions and products, and gradual type theory is preorder type theory with all of these.

$$\frac{A \text{ type} \qquad A' \text{ type}}{A \sqsubseteq A'} \qquad \frac{\Gamma \text{ ctx} \qquad \Gamma' \text{ ctx}}{\Phi : \Gamma \sqsubseteq \Gamma'} \qquad \frac{\Gamma \text{ ctx} \qquad A \text{ type}}{\Gamma \vdash t : A} \qquad \frac{\Phi : \Gamma \sqsubseteq \Gamma' \qquad \Gamma \vdash t : A}{A \sqsubseteq A' \qquad \Gamma' \vdash t' : A'}{\Phi \vdash t \sqsubseteq t' : A \sqsubseteq A'}$$

Figure 1 Judgment Presuppositions of Preorder Type Theory.

$$\frac{X \text{ base type}}{X \text{ type}} \qquad \frac{}{\cdot \text{ ctx}} \qquad \frac{\Gamma \text{ ctx} \qquad A \text{ type} \qquad x \notin \text{dom}(\Gamma)}{\Gamma, x : A \text{ ctx}} \qquad \frac{}{\Gamma, x : A, \Gamma' \vdash x : A}$$

Figure 2 Preorder Type Theory: Type and Term Structure.

Preorder Type Theory. Preorder type theory (PTT) has 6 judgments: types, contexts, type dynamism, dynamism contexts, terms and term dynamism. Their presuppositions (one is only allowed to make a judgment when these conditions hold) are presented in Figure 1, where A type and Γ ctx have no conditions. The types, contexts and terms (Figure 2) are structured as a standard call-by-name type theory. Terms are treated as intrinsically typed with respect to a context and an output type, contexts are ordered lists (this is important for our definition of dynamism context below). For bare preorder type theory, the only types are base types, and the only terms are variables and applications of uninterpreted function symbols (whose rule we omit). In the extended version [20], we give a precise definition of a *signature* specifying valid base types, function symbols, and type and term dynamism axioms. A substitution $\gamma : \Gamma' \vdash \Gamma$ is defined as usual as giving, for every typed variable in the output context $x : A \in \Gamma$, a term of that type relative to the input context $\Gamma' \vdash \gamma(x) : A$. Our term language supports a notion of substitution where if $\gamma : \Gamma' \vdash \Gamma$ and $\Gamma \vdash t : A$ then $\Gamma' \vdash t[\gamma] : A$.

Next, we discuss the new judgments of type dynamism, dynamism contexts, and term dynamism. A type dynamism judgment (Figure 3) $A \sqsubseteq B$ relates two well-formed types, and is read as "A is less dynamic than B". In preorder type theory, the only rules are reflexivity and transitivity, making type dynamism a preorder, and axioms from a signature (omitted).

The remaining rules in Figure 3 define *type dynamism contexts* Φ, which are used in the definition of term dynamism. While terms are indexed by a type and a typing context, term dynamism judgments $\Phi \vdash t \sqsubseteq t' : A \sqsubseteq A'$ are indexed by two terms $\Gamma \vdash t : A$ and $\Gamma' \vdash t' : A'$, such that $A \sqsubseteq A'$ (A is less dynamic than A') and Γ is less dynamic than Γ'. Thus, we require a judgment $\Phi : \Gamma \sqsubseteq \Gamma'$, which lifts type dynamism to contexts pointwise (for any $x : A \in \Gamma$, the corresponding $x' : A' \in \Gamma'$ satisfies $A \sqsubseteq A'$). This uses the structure of Γ and Γ' as ordered lists: a dynamism context $\Phi : \Gamma \sqsubseteq \Gamma'$ implies that Γ and Γ' have the same length and associates variables based on their order in the context, so that Φ is uniquely determined by Γ and Γ'; this is sufficient because of an admissible exchange rule for terms. We notate dynamism contexts to evoke a logical relations interpretation of term dynamism: under the conditions that $x_1 \sqsubseteq x_1' : A_1 \sqsubseteq A_1', \dots$ then we have that $t \sqsubseteq t' : B \sqsubseteq B'$.

The term dynamism judgment admits constructions (Figure 4) corresponding to both the structural rules of terms and the preorder structure of type dynamism, beginning from arbitrary term dynamism axioms (see the extended version [20] for a formal definition). First, there is a rule (TMPREC-VAR) that relates variables. Next there is a *compositionality* rule (TMPREC-COMP) that allows us to prove dynamism judgments by breaking terms down into components. We elide the definition of *substitution dynamism* $\Phi \vdash \gamma \sqsubseteq \gamma' : \Psi$, which is pointwise term dynamism. Last, we add an appropriate form of reflexivity (TMPREC-REFL)

$$\frac{}{A \sqsubseteq A} \qquad \frac{A \sqsubseteq B \qquad B \sqsubseteq C}{A \sqsubseteq C} \qquad \frac{}{\cdot : \cdot \sqsubseteq \cdot} \qquad \frac{\Phi : \Gamma \sqsubseteq \Gamma' \qquad A \sqsubseteq A'}{(\Phi, x \sqsubseteq x' : A \sqsubseteq A') : \Gamma, x : A \sqsubseteq \Gamma', x' : A'}$$

■ **Figure 3** Type and Context Dynamism.

$$\frac{x \sqsubseteq x' : A \sqsubseteq A' \in \Phi}{\Phi \vdash x \sqsubseteq x' : A \sqsubseteq A'} \ \text{TmPrec-Var}$$

$$\frac{\Phi \vdash t \sqsubseteq t' : A \sqsubseteq A' \qquad \Psi \vdash \gamma \sqsubseteq \gamma' : \Phi}{\Psi \vdash t[\gamma] \sqsubseteq t'[\gamma'] : A \sqsubseteq A'} \ \text{TmPrec-Comp}$$

$$\frac{\Gamma \vdash t : A \qquad \Phi : \Gamma \sqsubseteq \Gamma}{\Phi \vdash t \sqsubseteq t : A \sqsubseteq A} \ \text{TmPrec-Refl} \qquad \frac{\begin{array}{c}\Phi : \Gamma \sqsubseteq \Gamma' \vdash t \sqsubseteq t' : A \sqsubseteq A' \\ \Phi' : \Gamma' \sqsubseteq \Gamma'' \vdash t' \sqsubseteq t'' : A' \sqsubseteq A''\end{array}}{\Psi : \Gamma \sqsubseteq \Gamma'' \vdash t \sqsubseteq t'' : A \sqsubseteq A''} \ \text{TmPrec-Trans}$$

■ **Figure 4** Primitive Rules of Term Dynamism.

and transitivity (TmPrec-Trans) as rules, whose well-formedness depends on the reflexivity and transitivity of type dynamism. While the reflexivity rule is intuitive, the transitivity rule is more complex. Consider an example where $A \sqsubseteq A' \sqsubseteq A''$ and $B \sqsubseteq B' \sqsubseteq B''$:

$$\frac{x \sqsubseteq x' : A \sqsubseteq A' \vdash t \sqsubseteq t' : B \sqsubseteq B' \qquad x' \sqsubseteq x'' : A' \sqsubseteq A'' \vdash t' \sqsubseteq t'' : B' \sqsubseteq B''}{x \sqsubseteq x'' : A \sqsubseteq A'' \vdash t \sqsubseteq t'' : B \sqsubseteq B''}$$

In a logical relations interpretation of term dynamism, we would have relations $\sqsubseteq_{A,A'}$, $\sqsubseteq_{A',A''}$, $\sqsubseteq_{A,A''}$ and similarly for the B's, and the term dynamism judgment of the conclusion would be interpreted as "for any $u \sqsubseteq_{A,A''} u''$, $t[u/x] \sqsubseteq_{B,B''} t''[u''/x'']$". However, we could only instantiate the premises of the judgment if we could produce some middle u' with $u \sqsubseteq_{A,A'} u' \sqsubseteq_{A',A''} u''$. In such models, a middle u' *always* exists, because an implicit condition of the transitivity rule is that $\sqsubseteq_{A,A''}$ is the relation composite of $\sqsubseteq_{A,A'}$ and $\sqsubseteq_{A',A''}$ (the composite exists by type dynamism transitivity, and type dynamism witnesses are unique in PTT (thin in the semantics)). PTT itself does not give a term for this u', but the upcasts and downcasts in gradual type theory do (take it to be $\langle A' \hookleftarrow A \rangle u$ or $\langle A' \leftharpoondown A'' \rangle u''$).

Sometimes it is convenient to use the same variable name at the same type in both t and t', so we sometimes write $x : A$ in a dynamism context for $x \sqsubseteq x : A \sqsubseteq A$, and write Γ for $x_i \sqsubseteq x_i : A_i \sqsubseteq A_i$ for all $x_i : A_i$ in Γ. Similarly, we write A as the conclusion of a dynamism judgment for $A \sqsubseteq A$, so $\Gamma \vdash t \sqsubseteq t' : A$ means $\Gamma \sqsubseteq \Gamma \vdash t \sqsubseteq t' : A \sqsubseteq A$.

Gradual Type Theory. Preorder Type Theory gives us a simple foundation with which to build Gradual Type Theory in a modular way: we can characterize different aspects of gradual typing, such as a dynamic type, casts, and type errors separately.

We start by defining upcasts and downcasts, using type and term dynamism in Figure 5. Given that $A \sqsubseteq A'$, the upcast is a function from A to A' such that for any $t : A$, $\langle A' \hookleftarrow A \rangle t$ is the *least dynamic term of type A' that is more dynamic than t*. The UR rule can be thought of as the "introduction rule", saying $\langle A' \hookleftarrow A \rangle x$ is more dynamic than x, and then UL is the "elimination rule", saying that if some $x' : A'$ is more dynamic than $x : A$, then it

$$\frac{\Gamma \vdash t : A \qquad A \sqsubseteq A'}{\Gamma \vdash \langle A' \hookleftarrow A \rangle t : A'} \; \text{Upcast} \qquad\qquad \frac{\Gamma \vdash t : A' \qquad A \sqsubseteq A'}{\Gamma \vdash \langle A \leftarrow A' \rangle t : A} \; \text{Downcast}$$

$$\frac{A \sqsubseteq A'}{x \sqsubseteq x : A \sqsubseteq A \vdash x \sqsubseteq \langle A' \hookleftarrow A \rangle x : A \sqsubseteq A'} \; \text{UR} \qquad \frac{A \sqsubseteq A'}{x' \sqsubseteq x' : A' \sqsubseteq A' \vdash \langle A \leftarrow A' \rangle x' \sqsubseteq x' : A \sqsubseteq A'} \; \text{DL}$$

$$\frac{A \sqsubseteq A'}{x \sqsubseteq x' : A \sqsubseteq A' \vdash \langle A' \hookleftarrow A \rangle x \sqsubseteq x' : A' \sqsubseteq A'} \; \text{UL} \qquad \frac{A \sqsubseteq A'}{x \sqsubseteq x' : A \sqsubseteq A' \vdash x \sqsubseteq \langle A \leftarrow A' \rangle x' : A \sqsubseteq A} \; \text{DR}$$

$$\frac{A \sqsubseteq A'}{x : A \sqsubseteq x : A \vdash \langle A \leftarrow A' \rangle \langle A' \hookleftarrow A \rangle x \sqsubseteq x : A} \; \text{RetractAx} \qquad \frac{}{A \sqsubseteq \; ?} \qquad \frac{}{\Gamma \vdash \mho_A : A} \qquad \frac{\Phi : \Gamma \sqsubseteq \Gamma}{\Phi \vdash \mho_A \sqsubseteq t : A}$$

Figure 5 Upcasts, Downcasts, Dynamic Type and Type Error.

is more dynamic than $\langle A' \hookleftarrow A \rangle x$ – since $\langle A' \hookleftarrow A \rangle x$ is the *least* dynamic term with this property. The rules for projections are dual, ensuring that for $x' : A'$, $\langle A \leftarrow A' \rangle x'$ is the most dynamic term of type A that is less dynamic than x'. In fact, combined with the TmPrec-Trans rule, we can show that it has a slightly more general property: $\langle A' \hookleftarrow A \rangle x$ is not just less dynamic than any term of type A' more dynamic than x, but is less dynamic than any term of type A' *or higher*, i.e. of type $A'' \sqsupseteq A'$.

As we will discuss in Section 3, these rules allow us to prove that the pair of the upcast and downcast form a *Galois connection* (adjunction), meaning $\langle A' \hookleftarrow A \rangle \langle A \leftarrow A' \rangle t \sqsubseteq t$ and $t \sqsubseteq \langle A \leftarrow A' \rangle \langle A' \hookleftarrow A \rangle t$. However in programming practice, the casts satisfy the stronger condition of being a *Galois insertion*, in which the left adjoint, the downcast, is a *retract* of the upcast, meaning $t \sqsupseteq \sqsubseteq \langle A \leftarrow A' \rangle \langle A' \hookleftarrow A \rangle t$. We can restrict to Galois insertions by adding the *retract axiom* RetractAx. Most theorems of gradual type theory do not require it, though this axiom is satisfied in all models of preorder type theory in Section 6.

The remaining rules in Figure 5 define the dynamic type and type errors, which are also given a universal property in terms of type and term dynamism. The dynamic type is defined as the most dynamic type. The type error, written as $\mho$, is defined by the fact that it is a constant at every type A that is a least element of A. By transitivity, this further implies that $\mho_A \sqsubseteq t : A \sqsubseteq A'$ for any $A' \sqsupseteq A$.

Next we illustrate how simple *negative* types can be defined in preorder type theory. Figure 6 presents the rules for function types, while the product and unit types are analogous (see the extended version [20]). The type and term constructors are the same as those in the simply typed λ-calculus. Each type constructor extends type dynamism in the standard way [10, 31, 24]: every connective is *monotone* in every argument, including the function type. Due to the covariance of the function type, type dynamism is sometimes referred to as "naïve subtyping"; see 5 for a semantic intuition. For term dynamism, we add two classes of rules. First, there are congruence rules that "extrude" the term constructor rules for the type, which are like a "congruence of contextual approximation" condition. Next, the computational rules reflect the ordinary β, η equivalences as equi-dynamism: we write $\sqsupseteq \sqsubseteq$ to mean a rule exists in each direction (which requires that the types and contexts are also equi-dynamic).

We call the accumulation of all of these connectives *gradual type theory*. In the extended version [20], we define a GTT signature, which gives axioms for base types, function symbols, type dynamism, and term dynamism, which all may make use of the dynamic type, casts, type error, function types and product types, in addition to the rules of PTT.

$$\dfrac{A \sqsubseteq A' \quad B \sqsubseteq B'}{A \to B \sqsubseteq A' \to B'} \qquad \dfrac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : A \to B} \qquad \dfrac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B}$$

$$\dfrac{\Phi, x \sqsubseteq x' : A \sqsubseteq A' \vdash t \sqsubseteq t' : B \sqsubseteq B'}{\Phi \vdash \lambda x : A.t \sqsubseteq \lambda x' : A'.t' : A \to B \sqsubseteq A' \to B'} \qquad \dfrac{\Phi \vdash t \sqsubseteq t' : A \to B \sqsubseteq A' \to B' \quad \Phi \vdash u \sqsubseteq u' : A \sqsubseteq A'}{\Phi \vdash t\,u \sqsubseteq t'\,u' : B \sqsubseteq B'}$$

$$\dfrac{}{\Gamma \vdash (\lambda x : A.t)u \sqsupseteq\sqsubseteq t[u/x] : B} \qquad \dfrac{}{\Gamma \vdash t \sqsupseteq\sqsubseteq (\lambda x : A.t\,x) : A \to B \sqsubseteq A \to B}$$

Figure 6 Function Type.

3 Theorems and Constructions in Gradual Type Theory

In this section, we discuss some consequences of the simple axioms of gradual type theory. We show that almost every reduction in an operational presentation of call-by-name gradual typing, and many principles used in optimization of implementations, are justified by the universal property for casts in all types, the β, η rules, and the congruence rules for connectives and terms. Thus, the combination of graduality and η principles is a strong specification for gradual typing and considerably narrows the design space. We summarize these derivations in the following theorem:

▶ **Theorem 1.** *In Gradual Type Theory, all of the following are derivable whenever the upcasts, downcasts are well-formed.*
1. *Universal Property: Casts are unique up to $\sqsupseteq\sqsubseteq$.*
2. *Identity: $\langle A \hookleftarrow A \rangle t \sqsupseteq\sqsubseteq t$ and $\langle A \leftarrow A \rangle t \sqsupseteq\sqsubseteq t$.*
3. *Composition: $\langle A'' \hookleftarrow A \rangle t \sqsupseteq\sqsubseteq \langle A'' \hookleftarrow A' \rangle \langle A' \hookleftarrow A \rangle t$ and dually, $\langle A \leftarrow A'' \rangle t \sqsupseteq\sqsubseteq \langle A \leftarrow A' \rangle \langle A' \leftarrow A'' \rangle t$.*
4. *Function Cast Reduction: $\langle A' \to B' \hookleftarrow A \to B \rangle t \sqsupseteq\sqsubseteq \lambda x : A'.\langle B' \hookleftarrow B \rangle (t(\langle A \leftarrow A' \rangle x))$ and $\langle A \to B \leftarrow A' \to B' \rangle t \sqsupseteq\sqsubseteq \lambda x : A'.\langle B \leftarrow B' \rangle (t(\langle A' \hookleftarrow A \rangle x))$.*
5. *Product Cast Reduction: $\langle A'_0 \times A'_1 \hookleftarrow A_0 \times A_1 \rangle t \sqsupseteq\sqsubseteq (\langle A'_0 \hookleftarrow A_0 \rangle \pi_0 t, \langle A'_1 \hookleftarrow A_1 \rangle \pi_1 t)$ and $\langle A_0 \times A_1 \leftarrow A'_0 \times A'_1 \rangle t \sqsupseteq\sqsubseteq (\langle A_0 \leftarrow A'_0 \rangle \pi_0 t, \langle A_1 \leftarrow A'_1 \rangle \pi_1 t)$.*
6. *Adjunction: $t \sqsubseteq \langle A \leftarrow A' \rangle \langle A' \hookleftarrow A \rangle t$ and $\langle A' \hookleftarrow A \rangle \langle A \leftarrow A' \rangle t \sqsubseteq t$, for which the retract axiom is the converse.*
7. *Cast Congruence: $x \sqsubseteq y : A \sqsubseteq B \vdash \langle A' \hookleftarrow A \rangle x \sqsubseteq \langle B' \hookleftarrow B \rangle y : A' \sqsubseteq B'$ and $x' \sqsubseteq y' : A' \sqsubseteq B' \vdash \langle A \leftarrow A' \rangle x' \sqsubseteq \langle B \leftarrow B' \rangle y' : A \sqsubseteq B$.*
8. *Errors: $\langle A' \hookleftarrow A \rangle \mho_A \sqsupseteq\sqsubseteq \mho_{A'}$, and by the retract axiom $\langle A \leftarrow A' \rangle \mho_A \sqsupseteq\sqsubseteq .\mho_{A'}$.*
9. *Equi-dynamism implies isomorphism: If $A \sqsupseteq\sqsubseteq B$, then A is isomorphic to B.*

We present one example proof (most of the rest can be found in the extended version), and give some intuition for the others based on the defining properties of upcasts and downcasts as meets and joins. Part 1 says that this specification defines them *uniquely*, which we can prove by duplicating the rules for upcasts/downcasts and showing the two are $\sqsupseteq\sqsubseteq$. First, identity (2) and composition (3) are intuitive consequences that are sometimes operational reductions. Part 2 says the cast from a type to itself is the identity function and is easily justified by the specification: given $t : A$, t itself is the least dynamic element of A that is at least as dynamic as t. Part 3 says that if $A \sqsubseteq A' \sqsubseteq A''$ then casts between A, A'' factor through A'. This is important operationally, justifying the common rule $\langle A \to B \leftarrow ? \rangle t \mapsto \langle A \to B \leftarrow ? \to ? \rangle \langle ? \to ? \leftarrow ? \rangle t$ which says that casting to a function type

$$\cfrac{\cfrac{f, x \sqsubseteq x' \vdash f \sqsubseteq f : A \to B \sqsubseteq A \to B \qquad \cfrac{\cfrac{f, x \sqsubseteq x' \vdash x \sqsubseteq x' : A \sqsubseteq A'}{f, x \sqsubseteq x' \vdash x \sqsubseteq \langle A \leftarrow A' \rangle x' : A \sqsubseteq A}}{\cfrac{f, x \sqsubseteq x' \vdash fx \sqsubseteq f(\langle A \leftarrow A' \rangle x') : B \sqsubseteq B}{f, x \sqsubseteq x' \vdash fx \sqsubseteq \langle B' \leftharpoondown B \rangle (f(\langle A \leftarrow A' \rangle x')) : B \sqsubseteq B'}}}{\cfrac{\cfrac{f \sqsubseteq \lambda x. fx \qquad \cfrac{\lambda x. fx \sqsubseteq \lambda x' : A'. \langle B' \leftharpoondown B \rangle (f(\langle A \leftarrow A' \rangle x'))}{f \vdash f \sqsubseteq \lambda x' : A'. \langle B' \leftharpoondown B \rangle (f(\langle A \leftarrow A' \rangle x')) : A \to B \sqsubseteq A' \to B'}}{}}{f : A \to B \vdash \langle A' \to B' \leftharpoondown A \to B \rangle f \sqsubseteq \lambda x' : A'. \langle B' \leftharpoondown B \rangle (f(\langle A \leftarrow A' \rangle x')) : A' \to B'}}}$$

Figure 7 Function Upcast Implementation (one case).

first does the first order check to make sure t is a function, and then performs the checking of the function's behavior. More generally, it implies that casts from A to B commute over the dynamic type, e.g. $\langle ? \leftharpoondown B \rangle \langle B \leftharpoondown A \rangle x \sqsupseteq\sqsubseteq \langle ? \leftharpoondown A \rangle x$ – intuitively, if casts only perform checks, and do not change values, then a value's representation in the dynamic type should not depend on how it got there.

Next, consider the function contract reduction (4) (5 is similar). These equivalences are the standard "wrapping" implementations from [8, 7]: a function upcast uses the downcast on inputs and upcast on outputs and vice-versa for the downcast. This shows that the standard implementation is in fact the *unique* implementation to satisfy soundness and graduality. We present one of the cases for upcasts in Figure 7; the other 3 proofs are similar. First, we use the UL rule to reduce to showing the wrapping implementation is more dynamic than f itself. Then we use transitivity to η-expand f and then apply the λ congruence rule. Then we know the upcast $\langle B' \leftharpoondown B \rangle \cdot$ makes the term more dynamic and then apply function application congruence and use DR to show that the downcast of x' is still more dynamic than x. The other cases and the product cases follow by similar proofs.

Next, as mentioned previously, the adjunction property (6) shows that the upcast and downcast are a Galois connection with the upcast as the upper/left adjoint. This tells us that given $A \sqsubseteq A'$, the "round-trip" from A' down to A and back results in a less dynamic term and the other round-trip results in a more dynamic term. In programming practice, we expect the round trip from A to A' and back to be in fact an identity, as in the above retract axiom RETRACTAX. This theorem is the basis for our model (Section 6) where we *define* type dynamism as a pair of functions with these properties.

Next, it is important for proving the gradual guarantee [24] that all term constructors are congruences with respect to type and term dynamism. While for types like functions and products these are primitive rules, for casts congruence is derivable (7).

Error strictness (8) states that upcasts and downcasts are *strict* with respect to the type error $\mho$. The upcast preserves $\mho$ because it is a left/upper adjoint and therefore preserves colimits/joins like $\mho$. The proof that the downcast preserves $\mho$ is less modular, and uses the upcast, the retract axiom, and strictness of the upcast.

Finally, because types A and B in gradual type theory can be related both by type dynamism $A \sqsubseteq B$ and by functions $A \to B$, there are two reasonable notions of equivalence of types[3]. First, *equi-dynamism* $A \sqsupseteq\sqsubseteq B$ means $A \sqsubseteq B$ and $B \sqsubseteq A$. Second, isomorphism means there exist functions $f : A \to B$ and $g : B \to A$ such that $f \circ g \sqsupseteq\sqsubseteq (\lambda x : B.x)$ and $g \circ f \sqsupseteq\sqsubseteq (\lambda x : A.x)$. Part 9 gives an isomorphism for any equi-dynamic types.

---

[3] Corresponding to the two notions of isomorphism in double categories

The converse, isomorphic types are equi-dynamic, does not hold by design, because it does not match gradual typing practice. Gradually typed languages typically have *disjointness* of connectives as operational reductions; for example, disjointness of products and functions can be expressed by an axiom $\langle (C \times D) \leftarrow ?\rangle\langle ? \leftharpoonup (A \to B)\rangle x \sqsubseteq \mho$ which says that casting a function to a product errors. This axiom is incompatible with all isomorphic types being equi-dynamic, because a function type *can* be isomorphic to a product type (e.g. $X \to Y \cong (X \to Y) \times 1$), and for equi-dynamic types A and B, a cast $\langle B \leftarrow ?\rangle\langle ? \leftharpoonup A\rangle x$ should succeed, not fail (if it fails, then every term of A, B equal $\mho$; see the extended version). That is, disjointness axioms make equi-dynamism an intensional property of the representation of a type, and therefore stronger than isomorphism. Nonetheless, the basic rules of gradual type theory do not imply disjointness; in Section 6, we discuss a countermodel.

4 Categorical Semantics

Next, we define what a category-theoretic model of preorder and gradual type theory is, and prove that PTT/GTT are *internal languages* of these classes of models by proving soundness and completeness (i.e. initiality) theorems. This alternative axiomatic description of PTT/GTT is a useful bridge between the syntax and the concrete models presented in Section 6. The models are in *preorder categories*, which are categories internal to the category of preorders.[4] A preorder category is a category where the set of all objects and set of all arrows are each equipped with a preorder (a reflexive, transitive, but not necessarily anti-symmetric, relation). Furthermore the source, target, identity and composition functions are all *monotone* with respect to these orderings. A preorder category is equivalently a double category where one direction of morphism is thin. Intuitively, the preorder of objects represents types and type dynamism, while the preorder of morphisms represents terms and term dynamism, and we reuse the notation $\sqsubseteq$ for the orderings on objects and morphisms.

While the axioms of a preorder category are *similar to* the judgmental structure of preorder type theory, in a preorder category, morphisms have *one* source object and one target object, whereas in preorder type theory, terms have an entire *context* of inputs and one output. This is a standard mismatch between categories and type theories, and is often resolved by assuming that models have product types and using categorical products to interpret the context [13]. However, we will take a *multicategorical* view, in which our notion of model will axiomatize algebraically a notion of morphism with many inputs. Though for ordinary simple type theory the difference between the two is a matter of taste, for preorder type theory the difference is important when modeling term and specifically context dynamism. If a context is just a product of objects, then one context Γ should be less dynamic than another just when their product is, but in the syntax of our type theory, we only say a context is less dynamic than another when they are less dynamic point-wise. Since we allow for type dynamism axioms, this would mean that the syntax of context dynamism would be incomplete if it were interpreted this way. Instead, we give a multicategorical definition in which the notion of context dynamism in the model is also pointwise. Specifically, we define a model of preorder type theory to be a cartesian preorder multicategory, which is like a preorder multicategory that does not necessarily have true product *objects*, but whose morphisms' source can be a "virtual" product of objects, i.e. a context. For a general definition of multicategory that includes the notion we present, see [4]. In the extended version [20], we prove soundness and completeness of PTT for CPMs.

---

[4] To avoid confusion, these are not categories that happen to be preorders (thin categories) and these are not categories *enriched* in the category of preorders, where the hom-sets between two objects are preordered, but the objects are not.

▶ **Definition 2** (CPM). A cartesian preorder multicategory (CPM) $\mathbb{C}$ consists of a preordered set of "objects" $\mathbb{C}_0$, a preordered set of "multiarrows" $\mathbb{C}_1$, monotone functions "source" $s : \mathbb{C}_1 \to \mathrm{Ctx}(\mathbb{C})_0$, "target" $t : \mathbb{C}_1 \to \mathbb{C}_0$, "projection" $x : \mathrm{Ctx}(\mathbb{C})_0 \times \mathbb{C}_0 \times \mathrm{Ctx}(\mathbb{C})_0 \to \mathbb{C}_1$ and composition $\circ : \mathbb{C}_1 \times_{\mathrm{Ctx}(\mathbb{C})_0} \mathrm{Ctx}(\mathbb{C})_1 \to \mathbb{C}_1$. Here $\mathrm{Ctx}(\mathbb{C})_0$ is the set of lists of objects preordered pointwise, and a substitution $\gamma \in \mathrm{Ctx}(\mathbb{C})_1(\Gamma; B_1, \dots, B_n)$ consists of a multiarrow $\gamma(i) \in \mathbb{C}_1(\Gamma; B_i)$ for each $i \in 1, \dots, n$, also preordered pointwise, and with composition defined in the same way as syntactic substitutions. Additionally these satisfy associativity and unitality laws (see the extended version [20]).

Gradual Typing Structures. Next, we describe the additional structure on a CPM to model full gradual type theory: casts are modeled by an *equipment* [23], a dynamic type by a greatest object, and the type error by a least element of every hom-set.

▶ **Definition 3** (Gradual Structure on a CPM).
1. A CPM $\mathbb{C}$ is an *equipment* if for every $A \sqsubseteq B$, there exist morphisms $u_{A,B} \in \mathbb{C}(A, B)$ and $d_{A,B} \in \mathbb{C}(B, A)$ such that $u_{A,B} \sqsubseteq \mathrm{id}_B$ and $\mathrm{id}_A \sqsubseteq u_{A,B}$ and $d_{A,B} \sqsubseteq \mathrm{id}_B$ and $\mathrm{id}_A \sqsubseteq d_{A,B}$. An equipment is *coreflective* if also $d_{A,B} \circ u_{A,B} \sqsubseteq \mathrm{id}_A$.
2. A greatest object in a CPM $\mathbb{C}$ is a greatest element of the preorder of objects $\mathbb{C}_0$.
3. A CPM $\mathbb{C}$ has local bottoms if every hom set $\mathbb{C}(A_1, \dots, A_n; B)$ has a least element $\bot$ and for every substitution γ we have $\bot \circ \gamma \sqsupseteq\sqsubseteq \bot$.

Next, we define a *cartesian closed CPM*, which will model negative function and product types. We present the definition for function types in detail; a definition of a cartesian CPM (CPM with products) is in [20]. A *cartesian closed CPM* is a CPM with a choice of both cartesian and closed structure. Since all of the concrete models we consider are strict, we take a strict interpretation of naturality and $\beta\eta$, but this could likely be weakened.

▶ **Definition 4** (Closed CPM). A Closed CPM is a CPM $\mathbb{C}$ with a monotone function on objects $\to: \mathbb{C}_0^2 \to \mathbb{C}_0$ making for every pair of objects $X, Y \in \mathbb{C}$ an "exponential" object $X \to Y$ with a monotone function $\lambda : \mathbb{C}(\Gamma, X; Y) \to \mathbb{C}(\Gamma; X \to Y)$ that is *natural* in an appropriate sense, with a morphism $\mathrm{app} \in \mathbb{C}(X \to Y, X; Y)$ such that the function given by $f \mapsto \mathrm{app} \circ (f, x(X))$ is an inverse to λ (all up to equality).

In the extended version [20], we prove the following, where a *GTT category* is a CPM that is cartesian closed, a coreflective equipment and has a greatest object and local bottoms.

▶ **Definition 5** (Interpretation of Gradual Type Theory/Soundness). For any GTT signature Σ and GTT category $\mathbb{C}$ and interpretation $(\!|\cdot|\!) : \Sigma \to \mathbb{C}$ of the base types and function symbols in Σ such that all type and term dynamism axioms in Σ are true in $\mathbb{C}$, there is a compositional extension of $(\!|\cdot|\!)$ to an interpretation of all types and terms of GTT generated by Σ, such that all derivable type and term dynamism theorems are true in $\mathbb{C}$.

▶ **Theorem 6** (Completeness of GTT Category Semantics). *For any GTT signature Σ, for any GTT_Σ types A, B if for every interpretation $(\!|\cdot|\!) : \Sigma \to \mathbb{C}$, $[\![A]\!] \sqsubseteq [\![B]\!]$ holds, then $A \sqsubseteq B$ is derivable. For any GTT_Σ contexts $\Phi : \Gamma \sqsubseteq \Gamma'$, types $A \sqsubseteq A'$, and terms $\Gamma \vdash t : A$ and $\Gamma' \vdash t' : A'$, if for every interpretation $[\![t]\!] \sqsubseteq [\![t']\!]$, then $\Phi \vdash t \sqsubseteq t' : A \sqsubseteq A'$ derivable.*

As usual, the proof of completeness is by building a GTT category from the syntax such that the true dynamism theorems are precisely the derivable ones.

5   Semantic Contract Interpretation

As a next step towards constructing specific GTT categories, we define a general *contract construction* that provides a semantic account of the "contract interpretation" of gradual typing, which models a gradual type by a pair of casts. The input to our contract construction is a locally thin 2-category $\mathbb{C}$, whose objects and arrows should be thought of as the types and terms of a programming language, and each hom-set $\mathbb{C}(A, B)$ is ordered by an "approximation ordering", which is used to define term dynamism in our eventual model. We require each hom-set to have a least element (the type error), and the category to be cartesian closed (function and product types/contexts) in the strict sense of a 2-category whose underlying category is cartesian closed and where λ, application, pairing and projection are all functorial in 2-cells. The contract construction then "implements" gradual typing using the morphisms of the non-gradual "programming language" $\mathbb{C}$.

Coreflections.   To build a GTT model from $\mathbb{C}$, we need to choose an interpretation of type dynamism (the ordering on objects of the CPM) that induces appropriate casts, which we know by Theorem 1.6 must be Galois connections that satisfy the retract axiom. Such Galois connections are called Galois insertions (in order theory), coreflections (in category theory) and embedding-projection pairs (in domain theory). While we presented the retract axiom earlier as an $\sqsupseteq\sqsubseteq$, in all of our models the semantics of the composition $\langle A \twoheadleftarrow B \rangle \circ \langle B \hookleftarrow A \rangle$ is strictly equal to the identity so we will make a model using "strict" coreflections because it is slightly simpler. Since type dynamism judgments must induce a coreflection, we will construct a model where the semantics of a type dynamism judgment $A \sqsubseteq B$ is literally a coreflection. However, there can be many different coreflections between two objects of our 2-category $\mathbb{C}$, so this first step of our construction does not produce a preorder category, where type dynamism is an *ordering*, but rather a *double category*. Double categories generalize preorder categories in the same way that categories generalize preorders: the ordering on objects is generalized to proof-relevant data specifying a second class of *vertical morphisms*, and the ordering on terms becomes a notion of 2-dimensional "square" between morphisms. In the model we build from $\mathbb{C}$, the vertical morphisms will model type dynamism and be coreflections, while the *(horizontal)* morphisms of a preorder category will be arbitrary morphisms of $\mathbb{C}$ and model terms. We still require only double categories that are *locally thin*, in that there is at most one 2-cell filling in any square. Thus, the first step of our contract construction can be summarized as creating a double category that is an equipment with the retract property, i.e. a double category modeling upcasts and downcasts, a slight variation on a theorem in [23]:

▶ **Definition 7** (Equipment of Coreflections [23])**.** Given a 2-category $\mathbb{C}$ we construct a (double category) equipment $\mathrm{CoReflect}(\mathbb{C})$ as follows. Its object category has $\mathbb{C}_0$ as objects and coreflections in $\mathbb{C}$ as morphisms. Horizontal morphisms are given by morphisms in $\mathbb{C}$ and a 2-cell from $f : A \to B$ to $f' : A' \to B'$ along $(u_A, d_A) : A \lhd A'$ and $(u_B, d_B) : B \lhd B'$ is a 2-cell in $\mathbb{C}$ from $u_B \circ f$ to $f' \circ u_A$ or equivalently from $f \circ d_A$ to $d_B \circ f'$. From a vertical arrow (u, d), the upcast is u and the downcast is d.

As is well-known in domain theory, any mixed-variance functor preserves coreflections [32, 27], so the product and exponential functors of $\mathbb{C}$ extend to be functorial also in vertical arrows. This produces the classic "wrapping" construction familiar from higher-order contracts [8]:$(u, d) \to (u', d') = (d \to u', u \to d')$

Vertical Slice Category.   The double category CoReflect($\mathbb{C}$) is not yet a model of gradual type theory for two reasons. First, gradual type theory requires a dynamic type: every type should have a canonical coreflection into a specific type. Second, type dynamism in GTT is *proof-irrelevant*, because the rules do not track different witnesses of $A \sqsubseteq B$, but there may be different coreflections from A to B. It turns out that we can solve both problems at once by taking what we call the "vertical slice" category over an object $D \in$ CoReflect($\mathbb{C}$) that is rich enough to serve as a model of the dynamic type. In CoReflect($\mathbb{C}$)$/D$, the objects are not just an object A of $\mathbb{C}$, but an object *with* a vertical morphism into D, in this case a coreflection written $(u_A, d_A) : A \vartriangleleft D$.[5] Thus, gradual types are modeled as coreflections into the dynamic type, analogous to Scott's "retracts of a universal domain" [22]. Then a vertical arrow from $(u_A, d_A) : A \vartriangleleft D$ to $(u_B, d_B) : B \vartriangleleft D$ is a coreflection $(u_{A,B}, d_{A,B}) : A \vartriangleleft B$ that *factorizes* $u_A = u_B \circ u_{A,B}$ and $d_A = d_{A,B} \circ d_B$: this means the enforcement of A's type can be thought of as also enforcing B's type. Since upcasts are monomorphisms and downcasts are epimorphisms, this factorization is *unique* if it exists, so there is at most one vertical arrow between any two objects of CoReflect($\mathbb{C}$)$/D$. If we took the retract axiom as $\sqsupseteq\sqsubseteq$ rather than strict equality, then the factorization would only be *essentially unique*, i.e. any two factorizations would be equivalent; since our models are all strict, we defer exploring a weak variant to future work. Further, the identity coreflection $(\mathrm{id}, \mathrm{id}) : D \vartriangleleft D$ is a vertically greatest element since any morphism is factorized by the identity.

▶ **Definition 8** (Vertical Slice Category). Given any double category $\mathbb{E}$ and an object $D \in \mathbb{E}$, we can construct a double category $\mathbb{E}/D$ by defining $(\mathbb{E}/D)_0$ to be the slice category $\mathbb{E}_0/D$, a horizontal morphism from $(c : A \vartriangleleft D)$ to $(d : B \vartriangleleft D)$ to be a horizontal morphism from A to B in $\mathbb{E}$, and the 2-cells are similarly inherited from $\mathbb{E}$.

Next consider a cartesian closed structure on CoReflect($\mathbb{C}$)$/D$. The action of $\to$ (respectively $\times, 1$) on objects is given by composition of the action in CoReflect($\mathbb{C}$) $(u, d) \to (u', d')$ with an *arbitrary choice* of "encoding" of the "most dynamic function type" $(u_\to, d_\to) : (D \to D) \vartriangleleft D$. In most of the models we consider later, D is a sum and this coreflection simply projects out of the corresponding case, failing otherwise. This reflects the separation of the function contract into "higher-order" checking $(u, d) \to (u', d')$ and "first-order tag" checking $(u_\to, d_\to)$ that has been observed in implementations [10].

Finally, we construct a multicategory Multi($\mathbb{C}$) from the double category CoReflect($\mathbb{C}$)$/D$. A multiarrow $A_1, \ldots, A_n; B$ is given by a horizontal arrow $A_1 \times \ldots \times A_n; B$ in CoReflect($\mathbb{C}$)$/D$. The ordering $A \sqsubseteq A'$ is given by the vertical arrows $A \vartriangleleft A'$ of CoReflect($\mathbb{C}$)$/D$ (i.e. coreflections), which is lifted pointwise to contexts by the definition of a CPM. The ordering $f : (A_1, \ldots, A_n; B) \sqsubseteq g : (A'_1, \ldots, A'_n; B')$ is given by squares in CoReflect($\mathbb{C}$)$/D$ (using the action/monotonicity of $\times$ on the pointwise orderings $A_i \sqsubseteq A'_i$ of the context). Combining these constructions, we produce:

▶ **Theorem 9** (Contract Model of Gradual Typing). *If $\mathbb{C}$ is a locally thin cartesian closed 2-category with local $\bot s$, then for any object $D \in \mathbb{C}$ with chosen coreflections $c_\to : (D \to D) \vartriangleleft D$, $c_\times : (D \times D) \vartriangleleft D$, and $c_1 : 1 \vartriangleleft D$, then $(Multi(CoReflect(\mathbb{C})/id_d), c_\to, c_\times, c_1)$ is a GTT category.*

6   Concrete Models

Next, we produce some concrete models by instantiating Theorem 9.

---

[5]   We do not write $A \sqsubseteq D$ because coreflections are not a preorder.

There are two models based on domains that are operationally *inadequate* in that they identify the dynamic type error and diverging programs: term dynamism is given by the "definedness ordering" of domain theory. Both are based on the 2-category of domains (ω-cppos), continuous functions, and the domain ordering on functions, with different choices of universal domain. The first is merely a new presentation of Dana Scott's classical models of untyped λ-calculus, showing that Scott's model is already a model of gradual typing [22]. That is, we find the solution to the recursive domain equation $D \cong \mathbb{N}_\perp \oplus (D \times D) \oplus (D \to D)$ where $\oplus$ is the coalesced sum of domains. The classical technique for solving this equation naturally produce the required coreflections $(D \times D) \lhd D$ and $(D \to D) \lhd D$. The second is a variation where product and function types have overlapping representation, showing that the product and function types cannot be proven disjoint in gradual type theory. To do this, we construct a universal domain as a *product* of basic connectives rather than a sum: $D' \cong \mathbb{N}_\perp \times (D' \times D') \times (D' \to D')$. This is a kind of "coinductive"/"object-oriented" dynamic type: an element of the dynamic type responds to messages (given by the projections), and if it "doesn't implement" a message it returns $\perp$. Then $\langle (? \times ?) \leftarrow ? \rangle \langle ? \hookleftarrow (? \to ?) \rangle x \not\equiv \mho$ the domain has elements that are non-trivial in both the $D \times D$ and $D \to D$ positions.

Next, to produce a domain theoretic model that is adequate, we want a notion of domain that has, in addition to the definedness ordering needed for solving recursive domain equations, a *separate* notion of type error and error-approximation ordering. This can be accomplished using "pointed domain preorders", which are both domains and preorders with a least element in a compatible way. First, the diverging element must be *maximal* in the error ordering, so that they are sufficiently independent. Next, the error ordering must be an admissible relation so that the set of monotone functions is closed under limits.

▶ **Definition 10** ((Pointed) Domain Preorder). A domain preorder is a set X with two orderings $\leq$ and $\sqsubseteq$ such that $(X, \leq)$ is an ω-cppo with $\leq$-least element $\perp$ and $\sqsubseteq$ is a preorder closed under limits of $\leq$-ω-chains such that $\perp$ is $\sqsubseteq$-maximal. A continuous function of domain preorders is a function of the underlying sets that is continuous with respect to $\leq$ and monotone with respect to $\sqsubseteq$. A domain preorder is *pointed* if it has a $\sqsubseteq$-least element $\mho$.

The model is given by the 2-category of *pointed* domain preorders, with the $\sqsubseteq$ ordering, but to solve domain equations we use the category of all domain preorders. The 2-category with the domain ordering satisfies the criteria of [27] and so we can construct a suitable domain preorder by solving a similar equation to Scott's model: $D \cong \mathbb{N}_{\perp,\mho} \oplus (D_\mho \times D_\mho) \oplus (D_\mho \to D_\mho)$, where $D_\mho$ freely adjoins a $\sqsubseteq$-least element. Then the dynamic type is given by $D_\mho$, and we can construct coreflections (with respect to $\sqsubseteq$) $(D_\mho \times D_\mho) \lhd D_\mho$ and $(D_\mho \to D) \lhd D$: the downcast produces $\mho$ unless it is the $D \times D$ (respectively $D \to D$) case and the $1 \lhd D$ is the unique coreflection between those objects. We need the fact that $\perp$ is maximal to prove these functions are monotone, see the extended version for more details [20].

7 Related and Future Work

Our logic and semantics of type and term dynamism builds on the formulation introduced with the gradual guarantee in [24], but the rules of our system differ in two key ways. First, our system includes the β, η equivalences as equi-dynamism axioms, making term dynamism a more semantic notion. Second, we only allow casts that are either upcasts or downcasts (as defined by type dynamism), whereas their system allows for a more liberal "compatibility" condition. Accordingly our rules of dynamism for casts are slightly different, but where it makes sense, the rules of the two systems are interderivable.

Our semantic model of contracts as coreflections has precedent in much previous work, though we are the first to make precise the relationship to gradual typing's notions of type and term dynamism.

Henglein's work [10] on dynamic typing defines casts that are retracts to the dynamic type, introduced the upcast-followed-by-downcast factorization that we use here, and defines a syntactic rewriting relation similar to our term dynamism rules. Further they define a "subtyping" relation that is the same as type dynamism and characterize it by a semantic property analogous to the semantics of type dynamism in our contract model. The upcast-downcast factorization of an arbitrary cast is superficially similar to the work on triple casts in [26], which collapse a sequence of casts starting at A and ending at B into a downcast to $A \sqcap B$ followed by an upcast to B. But note that this factorization is opposite (downcast and then upcast), and the upcast-downcast factorization requires only a dynamic type, while the converse requires an appropriate middle type, similarly to *image factorization*. Moreover, [9] shows that the correctness of factorization through $A \sqcap B$ is not always possible.

Findler and Blume's work on contracts as *pairs of projections* [7] is also similar. There a contract is defined in an untyped language to be given by a *pair* of functions that divide enforcement of a type between the a "positive" component that checks the term and a "negative" component that checks the continuation, naturally supporting a definition of *blame* when a contract is violated. We give no formal treatment of blame in this paper, but our separation into upcasts and downcasts naturally supports a definition of blame analogous to theirs. In their paper, each component c is idempotent and satisfies $c \sqsubseteq \mathsf{id}$. Their work is fundamentally untyped so a direct comparison is difficult.

Recent work on interoperability in a (non-gradual) dependently typed language [5] defines several variations of Galois connections to serve as models of casts with different properties. This work validates their comments that ordinary monotone Galois connections serve as a model of the upcasts and downcasts associated to type precision.

There are two recent proposals for a more general theory of gradual typing: Abstracting Gradual Typing (AGT) [9] and the Gradualizer [3]. Broadly, their systems and ours are similar in that type dynamism and graduality are central and a gradually typed language is constructed from a statically typed language. Gradual type theory is quite different in that it is based on an axiomatic semantics, whereas both of theirs are based on operational semantics. As such our notion of gradual type soundness is stronger than theirs: we assert program equivalences whereas their soundness theorem is related to the syntactic type soundness theorem of the static language. Their systems also develop a *surface syntax* for gradually typed languages (including implicit casts and gradual type checking), whereas our logic here only applies to the *runtime semantics* of the language. Finally, AGT is based on abstract interpretation and uses a Galois insertion between gradual types and sets of static types, but we do not see a precise relationship to our use of coreflections.

Relative to this related work, we believe the axiomatic specification of casts via a universal property relative to dynamism is a new idea in gradual typing, as is our categorical semantics and the presentation of the contract interpretation as a model construction.

In this paper we have shown that the combination of soundness and graduality produces strong specifications for call-by-name gradual typing implementations. However so far we have only validated this by denotational semantics, and we plan to develop *operational models* of this kind of gradual type theory where term dynamism is modeled by a type of contextual approximation. We also will investigate extensions to richer languages. First, we would like to develop a similar theory for call-by-value gradual typing, as every gradually typed language in use today is call-by-value. We plan to build on existing work on categorical semantics and

universal properties of types in call-by-value [15, 28]. The combination of gradual typing and parametric polymorphism has proven quite complex [17, 19, 1, 12]. If we could show that the combination of graduality with parametricity has a unique implementation, as we have shown here for simple typing, it would provide a strong semantic justification for a design.

References

1 Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. Theorems for free for free: Parametricity, with and without types. In *International Conference on Functional Programming (ICFP)*, 2017.

2 Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. A theory of gradual effect systems. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 283–295, 2014.

3 Matteo Cimini and Jeremy G. Siek. Automatically generating the dynamic semantics of gradually typed languages. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 789–803, 2017.

4 G. S. H. Cruttwell and Michael A. Shulman. A unified framework for generalized multicategories. *Theory and Applications of Categories*, 24(21), 2009. `arXiv:arXiv:0907.2460`.

5 Pierre-Evariste Dagand, Nicolas Tabareau, and Éric Tanter. Foundations of Dependent Interoperability. working paper or preprint, 2017. URL: `https://hal.inria.fr/hal-01629909`.

6 Matthias Felleisen. On the expressive power of programming languages. *ESOP'90*, 1990.

7 Robby Findler and Matthias Blume. Contracts as pairs of projections. In *International Symposium on Functional and Logic Programming (FLOPS)*, 2006.

8 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP)*, pages 48–59, sep 2002.

9 Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2016.

10 Fritz Henglein. Dynamic typing: Syntax and proof theory. *Sci. Comput. Program.*, 22(3):197–230, 1994.

11 Atsushi Igarashi, Peter Thiemann, Vasco Vasconcelos, and Philip Wadler. Gradual session types. In *International Conference on Functional Programming (ICFP)*, 2017.

12 Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. On polymorphic gradual typing. In *International Conference on Functional Programming (ICFP), Oxford, United Kingdom*, 2017.

13 J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic.* Cambridge University Press, 1986.

14 Nico Lehmann and Éric Tanter. Gradual refinement types. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2017.

15 Paul Blain Levy. *Call-by-Push-Value.* Ph. D. dissertation, Queen Mary, University of London, London, UK, mar 2001.

16 Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws (sienna lectures). *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1983/1996.

17 Jacob Matthews and Amal Ahmed. Parametric polymorphism through run-time sealing, or, theorems for low, low prices! In *European Symposium on Programming (ESOP)*, mar 2008.

18 Eugenio Moggi. Notions of computation and monads. *Inform. And Computation*, 93(1), 1991.

19 Georg Neis, Derek Dreyer, and Andreas Rossberg. Non-parametric parametricity. In *International Conference on Functional Programming (ICFP)*, pages 135–148, sep 2009.

20 Max S. New and Daniel R. Licata. Call-by-name gradual type theory. *CoRR*, 2018. URL: http://arxiv.org/abs/1802.00061.

21 Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001.

22 Dana Scott. Data types as lattices. *Siam Journal on computing*, 5(3):522–587, 1976.

23 Michael Shulman. Framed bicategories and monoidal fibrations. *Theory and Applications of Categories*, 20(18):650–738, 2008.

24 Jeremy Siek, Micahel Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages*, SNAPL 2015, 2015.

25 Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop (Scheme)*, pages 81–92, 2006.

26 Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 365–376, 2010.

27 Michael B Smyth and Gordon D Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing*, 11(4), 1982.

28 Sam Staton and Paul Blain Levy. Universal properties of impure programming languages. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2013.

29 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: From scripts to programs. In *Dynamic Languages Symposium (DLS)*, pages 964–974, 2006.

30 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, 2008.

31 Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming (ESOP)*, pages 1–16, 2009.

32 Mitchell Wand. Fixed-point constructions in order-enriched categories. *Theoretical Computer Science*, 8(1):13–30, 1979.

33 Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, 2011.

Unique perfect matchings and proof nets

Lê Thành Dũng Nguyễn[1]

Département d'informatique, École normale supérieure, Paris Sciences et Lettres
45 rue d'Ulm, 75005 Paris, France
Université Paris 13, Sorbonne Paris Cité, LIPN, CNRS
99 avenue Jean-Baptiste Clément, 93430 Villetaneuse, France
nltd@nguyentito.eu
https://orcid.org/0000-0002-6900-5577

Abstract

This paper establishes a bridge between linear logic and mainstream graph theory, building previous work by Retoré (2003). We show that the problem of correctness for MLL+Mix proof nets is equivalent to the problem of uniqueness of a perfect matching. By applying matching theory, we obtain new results for MLL+Mix proof nets: a linear-time correctness criterion, a quasi-linear sequentialization algorithm, and a characterization of the sub-polynomial complexity of the correctness problem. We also use graph algorithms to compute the dependency relation of Bagnol et al. (2015) and the kingdom ordering of Bellin (1997), and relate them to the notion of blossom which is central to combinatorial maximum matching algorithms.

1 Introduction

One of the major novelties introduced at the birth of linear logic [13] was a representation of proofs as *graphs*, instead of trees as in natural deduction or sequent calculus. A distinctive property of these *proof nets* is that checking that a proof is correct cannot be done merely by a local verification of inference steps: among the graphs which locally look like proof nets, called *proof structures*, some are invalid proofs. Hence the problem of *correctness*: given a proof structure, is it a real proof net?

A lot of work has been devoted to this decision problem, and in the case of the multiplicative fragment of linear logic (MLL), whose proof nets are the most satisfactory, it can be considered solved from an algorithmic point of view. Indeed, Guerrini [14] and Murawski and Ong [22] have found linear-time tests for MLL correctness; the problem has also been shown to be NL-complete by Jacobé de Naurois and Mogbil [17]. Both the linear-time algorithms

---

we mentioned also solve the corresponding search problem: computing a *sequentialization* of a MLL proof net, i.e. a translation into sequent calculus.

However, for MLL extended with the *Mix rule* [10] (MLL+Mix), the precise complexity of deciding correctness has remained unknown (though a polynomial-time algorithm was given by Danos [7]). Thus, one of our goals in this paper is to study the following problems:

▶ **Problem** (MixCorr)**.** Given a proof structure π, is it an MLL+Mix proof net?

▶ **Problem** (MixSeq)**.** Reconstruct a sequent calculus proof for an MLL+Mix proof net.

It turns out that a *linear-time* algorithm for MixCorr follows immediately from already known results. The key is to use a construction by Retoré [26, 27] to reduce it to the problem of *uniqueness of a given perfect matching*, which can be solved in linear time [11]:

▶ **Problem** (UniquenessPM)**.** Given a graph G, together with a *perfect matching M of G*, is M the only perfect matching of G? Equivalently, is there no *alternating cycle* for M?

This brings us to the central idea of this paper: *from the point of view of algorithmics, MLL+Mix proof nets and unique perfect matchings are essentially the same thing.* This allows us to apply matching theory to the study of proof nets, leading to several new results. Indeed, one would expect graph algorithms to be of use in solving problems on proof structures, since they are graphs! But for this purpose, a bridge between the theory of proof nets and mainstream graph theory is needed, whereas previous work on the former mostly made use of "homemade" objects such as *paired graphs* (an exception being Murawski and Ong's use of *dominator trees*). By building on Retoré's discovery of a connection with perfect matchings, this paper proposes such a bridge.

Plan of the paper and contributions.   First, we establish our equivalence by giving a translation from proof structures to graphs equipped with perfect matchings and vice versa (§3). In the first direction, instead of reusing Retoré's construction, we propose an alternative having better properties with respect to sequentialization.

As already mentioned, we give the first linear-time algorithm for MixCorr (§4.1). As for its sub-polynomial complexity (§4.2), we show that MixCorr is in randomized NC and in quasi-NC (informally, NC is the class of problems with efficient *parallel* algorithms). On the other hand, we have a sort of hardness result: if MixCorr were in NC – in particular, if it were in NL, as for MLL without Mix – this would imply a solution to a long-standing conjecture concerning the related *unique perfect matching* problem:

▶ **Problem** (UniquePM [19, 11, 15])**.** Given a graph G, determine whether it admits exactly one perfect matching and, if so, find this matching.

We then turn to the sequentialization problem, for which we provide a graph-theoretic reformulation, and an algorithm for this reformulation. This gives us a *quasi-linear* time[2] solution to MixSeq (§5); to our knowledge, this beats previous algorithms for MixSeq.

As a demonstration of our matching-theoretic toolbox, we also show how to compute some information on the set of *all* sequentializations, namely Bellin's *kingdom ordering* [4] of the links of a MLL+Mix proof net (rediscovered by Bagnol et al. [1] under the name of *order*

---

[2] More precisely, $O(n(\log n)^2(\log \log n)^2)$ time. Both this and our quasi-NC algorithms rely on very recent advances, respectively on dynamic bridge-finding data structures [16] and on the perfect matching existence problem [28]. Any further progress on these problems would lead to an improvement of our complexity bounds.

(a) Two perfect matchings of the same graph.   **(b)** A graph with a unique perfect matching.
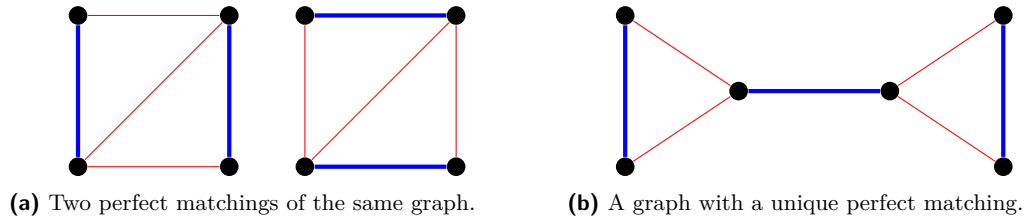
■ **Figure 1** Examples of perfect matchings. The edges in the matchings are thick and blue.

of introduction). We give a polynomial time and a quasi-NC algorithm (§6.1), both relying on an effective characterization of this ordering. By rephrasing this characterization (§6.2), we get a purely graph-theoretic new theorem of independent interest about objects which play a major role in matching algorithms, namely *blossoms* [9].

2   Preliminaries

Graph-theoretic terminology.   By default, "graph" refers to an *undirected* graph. Our *paths* and *cycles* are not allowed to contain repeated vertices[3]; we will sometimes identify them with their sets of edges (which characterize them) and apply set operations on them. A *bridge* of a graph is an edge whose removal increases the number of connected components.

For directed graphs, the notion of connectedness we consider is *weak connectedness*, i.e. connectedness of the graph obtained by forgetting the edge directions. A *predecessor* (resp. *successor*) of a vertex is the source (resp. target) of some incoming (resp. outgoing) edge.

Complexity classes.   We refer to [17, §1.4] for the logarithmic space classes L and NL and to [6] for the class AC^0 of constant-depth circuits. The class NC^k (resp. quasi-NC^k [3]) consists of the problems which can be solved by a uniform family of circuits of depth $O(\log^k n)$ and polynomial (resp. quasi-polynomial, i.e. $2^{O(\log^c n)}$) size; $\mathsf{NC} = \bigcup_k \mathsf{NC}^k$ and quasi-$\mathsf{NC} = \bigcup_k$ quasi-NC^k. It is well-known that $\mathsf{AC}^0 \subseteq \mathsf{NC}^1 \subseteq \mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{NC}^2 \subseteq \mathsf{NC} \subseteq \mathsf{P}$.

2.1   Perfect matchings, alternating cycles and sequentialization

▶ **Definition 2.1.** Let $G = (V, E)$ be a graph. A *matching* (resp. *perfect matching*) M in G is a subset of E such that every vertex in V is incident to *at most one* (resp. *exactly one*) edge in M. An *alternating path* (resp. *cycle*) for M is a path (resp. cycle) where, for every pair of consecutive edges, one of them is in the matching and the other one is not.

Testing the existence of a perfect matching in a graph – or, more generally, finding a maximum cardinality matching – is one of the central computational problems in graph theory. Combinatorial maximum matching algorithms, starting[4] with Edmonds's *blossom algorithm* [9][5], use alternating paths to iteratively increase the size of the matching; similarly, alternating cycles are important for the problems UNIQUENESSPM and UNIQUEPM because they witness the *non-uniqueness* of perfect matchings.

---

[3] This choice of terminology is common, see e.g. [2, §1.4].
[4] Note that the problem was solved long before in the special case of bipartite graphs. In fact, a solution for this case was found in Jacobi's posthumous papers.
[5] This paper is one of the first to propose defining efficient algorithms as polynomial-time algorithms; it also contributed to the birth of the field of polyhedral combinatorics.

▶ **Lemma 2.2** (Berge). *Let G be a graph and M be a perfect matching of G. Then if $M' \neq M$ is a perfect matching, the symmetric difference $M \triangle M'$ is a vertex-disjoint union of cycles, which are alternating for both M and M'. Conversely, if C is an alternating cycle for M, then $M \triangle C$ is another perfect matching.*

As an example, consider Figure 1a. The matching on the left admits an alternating cycle, the outer square; by taking the symmetric difference between this matching and the set of edges of the cycle, one gets the matching on the right. Conversely, the symmetric difference between both matchings (which, in this case, is their union) is the square. Note also that in Figure 1b, there is no alternating cycle because vertex repetitions are disallowed.

Another approach to finding perfect matchings, using linear algebra, was initiated by Lovász [20] and leads to a *randomized* NC algorithm by Mulmuley et al. [21]. Recently, Svensson and Tarnawski have shown that this algorithm can be derandomized to run in deterministic quasi-NC [28].

There is also a considerable body of purely mathematical work on matchings, starting from the 19th century. Let us mention for our purposes a result dating from 1959.

▶ **Theorem 2.3** (Kotzig [18]). *Let G be a graph. Suppose that G admits a unique perfect matching M. Then M contains a bridge of G.*

As remarked by Retoré [27], Kotzig's theorem leads to an inductive characterization of the set of graphs equipped with a unique perfect matching.

▶ **Theorem 2.4** (Sequentialization for unique perfect matchings [27]). *The class $\mathcal{UPM}$ of graphs equipped with an unique perfect matching is inductively generated as follows:*
- *The empty graph (with the empty matching) is in $\mathcal{UPM}$.*
- *The disjoint union of two non-empty members of $\mathcal{UPM}$ is in $\mathcal{UPM}$.*
- *Let $(G = (V, E), M \subseteq E) \in \mathcal{UPM}$ and $(G' = (V', E'), M' \subseteq E') \in \mathcal{UPM}$, with V and V' disjoint. Let $U \subseteq V$, $U' \subseteq V'$ such that $U \neq \emptyset$ (resp. $U' \neq \emptyset$) unless G (resp. G') is the empty graph, and let x, x' be two fresh vertices not in V nor V'. Then $(G'' = (V'', E''), M'' \subseteq E'') \in \mathcal{UPM}$, where*
  - $V'' = V \cup V' \cup \{x, x'\}$
  - $E'' = E \cup E' \cup \{(x, x')\} \cup (U \times \{x\}) \cup (U' \times \{x'\})$
  - $M'' = M \cup M' \cup \{(x, x')\}$

▶ *Remark.* By relaxing the non-emptiness condition on U and U', the disjoint union operation becomes unnecessary; this is actually the original statement [27, Theorem 1].

The inspiration for the above theorem comes from linear logic: it is a graph-theoretic version of the sequentialization theorems for proof nets, with Kotzig's theorem being analogous to the "splitting lemmas" which appear in various proofs of sequentialization.

2.2   Proof structures, proof nets and the correctness criterion

A proof structure is some kind of graph-like object made of "nodes" (or "formulae") and "links", with the precise definition varying in the literature. Since our aim is to apply results from graph theory, it will be helpful to commit to a representation of proof structures as graphs. (We write $\deg^-$ for the indegree and $\deg^+$ for the outdegree of a vertex.)

▶ **Definition 2.5.** A *proof structure* is a non-empty directed acyclic multigraph (V, A) with a labeling of the vertices $l : V \to \{\mathtt{ax}, \otimes, \mathbin{⅋}\}$ such that, for $v \in V$:
- if $l(v) = \mathtt{ax}$, then $\deg^-(v) = 0$ and $\deg^+(v) \leq 2$,
- if $l(v) \in \{\otimes, \mathbin{⅋}\}$, then $\deg^-(v) = 2$ and $\deg^+(v) \leq 1$.
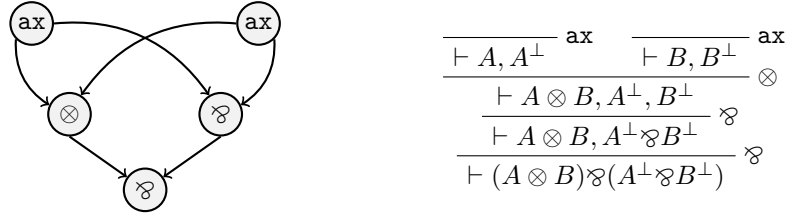
Figure 2 A proof net (left) and its sequentialization (right), written as a sequent calculus proof. Edges are usually labeled by the MLL formulae appearing in the sequentialization; since we focus on the combinatorics of proof structures and not on their logical meaning, we omit them here.

$$\frac{}{\vdash A, A^\perp}\text{(ax-rule)} \quad \frac{\vdash \Gamma, A \quad \vdash B, \Delta}{\vdash \Gamma, A \otimes B, \Delta}\text{($\otimes$-rule)} \quad \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \parr B}\text{($\parr$-rule)} \quad \frac{\vdash \Gamma \quad \vdash \Delta}{\vdash \Gamma, \Delta}\text{(Mix rule)}$$

Figure 3 Rules for the MLL+Mix sequent calculus; note the correspondence with Definition 2.6.

Vertices of a proof structure will also be called *links*. A *terminal link* is a link with outdegree 0. A *sub-proof structure* is a vertex-induced subgraph which is a proof structure.

▶ **Definition 2.6.** The set of *MLL proof nets* is the subset of proof structures inductively generated by the following rules:

- **ax-rule**: a proof structure with a single ax-link is a proof net.
- **⊗-rule**: if N and N' are proof nets, u is a link of N and v is a link of N', then taking the disjoint union of N and N', adding a new ⊗-link w, an edge from u to w and an edge from v to w gives a proof net, as long as the resulting graph is a proof structure (i.e. the degree constraints are satisfied).
- **⅋-rule**: if N is a proof net and u, v are links of N, then adding a new ⅋-link w, an edge from u to w and an edge from v to w gives a proof net, with the same proviso as above.

The set of *MLL+Mix proof nets* is inductively generated by the above rules together with the **Mix rule**: if N and N' are proof nets, their disjoint union is a proof net.

A proof structure is said to be *correct* if it is a MLL+Mix proof net.

▶ Remark. As with any inductively defined set, membership proofs for the set of MLL (resp. MLL+Mix) proof nets may be presented as inductive derivation trees, which are isomorphic to the usual *sequent calculus proofs* of MLL (resp. MLL+Mix): see Figure 2 for an example, and Figure 3 for the inference rules of the sequent calculus.

▶ Remark. The proof structures and proof nets defined here are *cut-free*. This restriction is without loss of generality, since cut link has exactly the same behavior as a terminal ⊗-link with respect to correctness and sequentialization.

To tackle the problem of correctness, it is useful to have non-inductive characterizations of proof nets, called *correctness criteria*, at our disposal. Many of them are formulated using the notion of *paired graphs*. We will state a criterion first discovered by Danos and Regnier for MLL [8] and extended to MLL+Mix by Fleury and Retoré [10].

▶ **Definition 2.7.** A *paired graph* consists of an undirected graph $G = (V, E)$ and a set $\mathcal{P}$ of unordered pairs of edges such that:

- if $\{e, f\} \in \mathcal{P}$, then e and f have a vertex in common;
- the pairs are disjoint: if $p, p' \in \mathcal{P}$ and $p \neq p'$, then $p \cap p' = \emptyset$.

When $\{e, f\} \in \mathcal{P}$, the edges e and f are said to be *paired*.

A *switching* of this paired graph is a spanning subgraph of G which intersects each pair of $\mathcal{P}$ exactly once. A *feasible cycle* is a cycle which intersects each pair of $\mathcal{P}$ at most once.

▶ Remark. Equivalently, feasible cycles are cycles which exist in some switching.

▶ **Definition 2.8.** Let π be a proof structure. Its *correctness graph* $C(\pi)$ is the paired graph obtained by forgetting the directions of the edges and the labels of the vertices in π, and pairing together two edges when their targets[6] are the same $\invamp$-link.

A *feasible cycle* in π is a sequence of edges of π whose image in $C(\pi)$ is a feasible cycle.

▶ **Theorem 2.9** (Danos–Regnier correctness criterion). *π is a MLL (resp. MLL+Mix) proof net if and only if all the switchings of $C(\pi)$ are trees (resp. forests).*

▶ Remark. Equivalently, π is a MLL+Mix proof net iff it contains no feasible cycle.

The above is usually called a *sequentialization theorem*: it means that a proof structure which satisfies the correctness criterion admits a sequent calculus derivation.

The analogy with Theorem 2.4 is that proof nets are to proof structures what *unique* perfect matchings are to perfect matchings. The next section is dedicated to formalizing this analogy into an equivalence.

3 An equivalence through mutual reductions

We will now see how to turn a proof structure into a graph equipped with a perfect matching, in such a way that feasible cycles become alternating cycles, and vice versa.

Such a translation from proof structures to perfect matchings was first proposed by Retoré [27], under the name of *R&B-graphs*. However, we would like to deduce Theorem 2.9 as an immediate corollary of sequentialization for unique perfect matchings (Theorem 2.4), which is not possible with R&B-graphs – instead, one must resort to a proof of induction using Kotzig's theorem (Theorem 2.3), see [26, §2.4]. Thus, we propose here our own *graphification* construction. We also define the *proofification* construction, going from perfect matchings to proof structures.

▶ Remark. The nature of the object corresponding to a matching edge in a proof structure will vary depending on the translation considered: for graphifications, they correspond to links, whereas in the case of proofifications, they are translated into $\otimes$-links (and for R&B-graphs, they correspond to edges or terminal links).

Thus, by taking the proofification of a graphification of a proof structure, one gets a different proof structure, with the $\mathtt{ax}$-links and $\invamp$-links of the former being sent to $\otimes$-links of the latter (see Figure 4 for an example). It is unclear whether this transformation has any meaning in terms of linear logic; in particular it does not preserve correctness for MLL without Mix.

3.1 From proof structures to perfect matchings

▶ **Definition 3.1.** Let π be a proof structure and L be its set of links. The *graphification* of π is the graph $G = (V, E)$ equipped with a perfect matching $M \subseteq E$ with
- the matching edges corresponding to the links: $V = \bigcup_{l \in L} \{a_l, b_l\}$, $M = \{(a_l, b_l) \mid l \in L\}$,
- and the remaining edges in $E \setminus M$ reflect the incoming edges of the $\otimes$-links and $\invamp$-links, as specified by Figure 5a.

---

[6] That is, the targets of the directed edges in π they come from.

(a) Before graphification. The left ⊗-link and the right ⊗-link of (b) correspond to the `ax`-links here.

(b) After proofification.
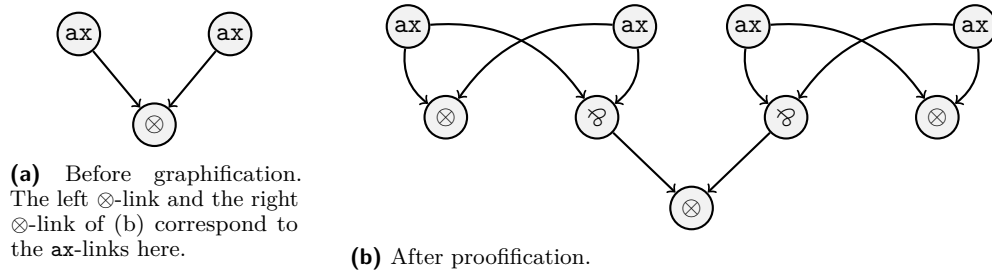
Figure 4 Composing graphification and proofification: as we will see, the graphification of the proof net of (a) is the graph of Figure 1b, and, in turn, the proofification of Figure 1b is (b).

(a) Translation rules for sets of incoming edges.

(b) Graphification of the proof structure of Figure 2
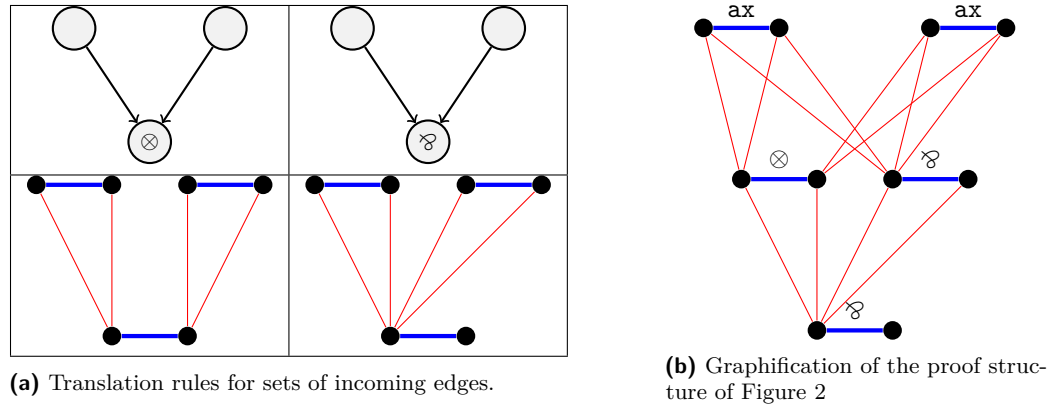
Figure 5 The graphification construction.

Figure 5b shows an example of this construction. As another example, Figure 1b is the graphification of Figure 4a.

▶ **Proposition 3.2** (Graphification-based correctness criterion). *A proof structure satisfies the Danos–Regnier criterion for MLL+Mix if and only if the perfect matching of its graphification is unique.*

In the case of R&B-graphs, there is an actual bijection between the feasible cycles of a proof structure and the alternating cycles of its R&B-graph. That said, the main technical advantages of graphifications over R&B-graphs are summarized by the following properties.

▶ **Lemma 3.3.** *Let π be a proof structure with graphification (G, M) and l be a link of π such that $(a_l, b_l) \in M$ is a bridge of G. Then l is a terminal link in π, and if l is a ⊗-link, then removing l from π disconnects its predecessors.*

▶ **Theorem 3.4.** *Let π be a proof structure and (G, M) be its graphification. There is a bijection between the sequent calculus proofs corresponding to π (if any) and the sequential-izations (i.e. the derivation trees for the inductive definition of Theorem 2.4) of (G, M) (if any), through which occurrences of Mix rules correspond to disjoint unions and conversely.*

In particular, π is a MLL+Mix proof net if and only (G, M) admits a sequentialization, that is, according to Theorem 2.4, if and only if M is the only perfect matching of G. Proposition 3.2 tells us that this is equivalent to π satisfying the Danos–Regnier acyclicity criterion. Therefore, this criterion characterizes MLL+Mix proof nets: as we wanted, we just proved the sequentialization theorem for MLL+Mix (Theorem 2.9).
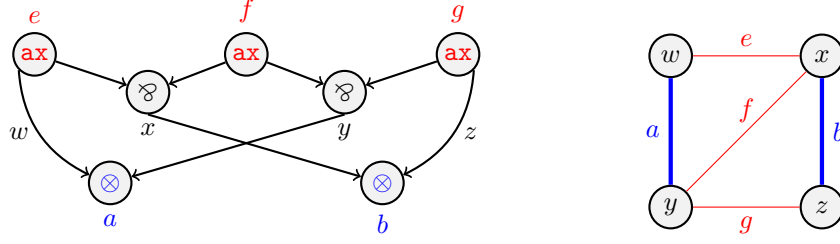
Figure 6 The prooffication of the graph of Figure 1a.

3.2 From perfect matchings to proof structures

The translation we present below involves "k-ary $\invamp$-links". When $k > 1$, these are just binary trees of $k - 1$ $\invamp$-links (correctness is independent of the choice of binary tree: semantically, this is associativity of $\invamp$) with k leaves (incoming edges) and a single root (outgoing edge); the $k = 1$ case corresponds to a single edge and no link.

▶ **Definition 3.5.** Let $G = (V, E)$ be a graph and M be a perfect matching of G. We define the *prooffication* of (G, M) as the proof structure π built as follows:

- For each non-matching edge $e = (u, v) \in E \setminus M$, we create an `ax`-link $\mathtt{ax}_e$ whose two outgoing edges we will call $A_{u,v}$ and $A_{v,u}$.
- For each vertex $u \in V$, if $\deg(u) > 1$, we add a k-ary $\invamp$-link with $k = \deg(u) - 1$, whose incoming edges are the $A_{u,v}$ for all neighbors v of u such that $(u, v) \notin M$, and we call its outgoing edge B_u. If $\deg(u) = 1$, we add an `ax`-link calling one of its outgoing edges B_u.
- For each matching edge $(u, v) \in M$, we add an $\otimes$-link whose incoming edges are B_u and B_v. These $\otimes$-links are the terminal links of π.

See Figure 6 for an annotated example of prooffication. The reader may also check that the proof net in Figure 4b is the prooffication of the graph in Figure 1b.

▶ **Proposition 3.6.** *Let G be a graph and M be a perfect matching of G. The alternating cycles for M in G are in bijection with the feasible cycles in the prooffication of (G, M).*

▶ **Proposition 3.7.** *Let G be a graph with a unique perfect matching M and let π be the prooffication of (G, M). A matching edge $e \in M$ is a bridge of G if and only if its corresponding $\otimes$-link is introduced by the last rule of some sequentialization of π.*

However, unlike the case of graphifications, this does not give us a bijection between the sequentializations of a unique perfect matching and those of its prooffication.

4 On the complexity of MLL+Mix correctness

Through the translations of the previous section, MLL+Mix proof *nets* become *unique* perfect matchings and conversely: these translations provide *reductions* between the problems MixCorr and UniquenessPM, allowing us to draw complexity-theoretic conclusions on proof nets from known results in graph theory. We first look at the time complexity of MixCorr, then turn to its complexity under constant-depth (AC^0) reductions.

4.1 A linear-time algorithm

Since graphifications (§3.1) can be computed in linear time, and UniquenessPM can also be decided in linear time [11, §3], we immediately get:

▶ **Theorem 4.1.** MixCorr *can be decided in linear time.*

▶ Remark. By using the "Euler–Poincaré lemma" [1] to count the uses of the Mix rule in a proof net, this also allows us to decide the correctness of a proof structure for MLL without Mix in linear time. Our decision procedure has the advantage of being simpler to describe than the previously known linear-time algorithms for MLL correctness [14, 22].

That said, this apparent simplicity is due to our use of the algorithm of Gabow et al. [11] as a black box. Looking inside the black box reveals, for instance, that it uses the *incremental tree set union* data structure of Gabow and Tarjan [12], which is also a crucial ingredient of the above-mentioned previous algorithms.

▶ Remark. This algorithm for UniquenessPM relies on the technique of *blossom shrinking* pioneered by Edmonds [9], a kind of graph contraction which may remind us of the *contractibility* correctness criterion [7] for MLL without Mix. Indeed, there exists a formal connection: a rewrite step of *big-step contractibility* [1] corresponds, when translated to graphifications, to contracting a blossom. However, not all blossoms are redexes for big-step contractibility.

4.2 Characterizing the sub-polynomial complexity

For MLL proof nets without Mix, correctness is known to be NL-complete under AC^0 reductions thanks to the Mogbil–Naurois criterion [17]. What about MLL+Mix? Since the reductions of §3 can be computed in constant depth, we have:

▶ **Theorem 4.2.** MixCorr *and* UniquenessPM *are equivalent under* AC^0 *reductions.*

Thus, it will suffice to study the complexity of UniquenessPM. Let us start with a positive result, using the parallel algorithms for finding a perfect matching mentioned in §2.1.

▶ **Proposition 4.3.** UniquenessPM *is in* randomized NC *and in* deterministic quasi-NC.

Proof. Let $G = (V, E)$ be a graph and M be a perfect matching of G. M is *not* unique if and only if, for some $e \in M$, the graph $G_e = (V, E \setminus \{e\})$ has a perfect matching. To test the uniqueness of M, run the $|M|$ parallel instances, one for each G_e, of a randomized NC [21] or deterministic quasi-NC [28] algorithm for deciding the existence of a perfect matching, and compute the disjunction of their answers in AC^0. ◀

Being in quasi-NC is a much weaker[7] result than being in NL. But as we shall now see, even showing that UniquenessPM is in NC (recall that NL ⊂ NC) would be a major result. It would answer in the affirmative the following conjecture dating back from the 1980's:

▶ **Conjecture 4.4** (Lovász[8])**.** UniquePM *is in* NC.

Indeed, the following shows that UniquenessPM ∈ NC ⇒ UniquePM ∈ NC (and the converse follows from the definitions).

▶ **Proposition 4.5.** *There is a* NC^2 *reduction from* UniquePM *to* UniquenessPM.

---

[7] In fact, one can show that NL ⊊ NSPACE$(O(\log^{3/2} n)) \subseteq$ quasi-NC^3, and the latter is where Svensson and Tarnawski's analysis puts finding a perfect matching.

[8] The conjecture is attributed to Lovász by a paper by Kozen et al. [19] which claims to solve it. But Hoang et al. [15] note that "this was later retracted in a personal communication by the authors". Still, the proposed solution works for bipartite graphs.

Proof. This is a consequence of a NC^2 algorithm by Rabin and Vazirani [25, §4] which, given a graph G, computes a set of edges M such that if G admits a unique perfect matching, then M is this matching. Starting from any graph G, run this algorithm and test whether its output is a perfect matching. If not, then G does not admit a unique perfect matching; if it is, then G is a positive instance of UNIQUEPM if and only if (G, M) is a positive instance of UNIQUENESSPM. ◀

To sum up these results about UNIQUENESSPM, which apply to MIXCORR:

▶ **Theorem 4.6.** MIXCORR *is in randomized* NC *and in deterministic* quasi-NC*; it is in deterministic* NC *if and only if Conjecture 4.4 is true.*

5 Sequentializing MLL+Mix proof nets

In §4.1, we managed to solve MLL+Mix correctness in linear time, matching the known time complexity for MLL correctness. But the algorithms for MLL correctness still have an advantage: they can compute a sequentialization in linear time, whereas we only have a decision procedure for MIXCORR which returns a yes/no answer[9]. We do not know how to compute MLL+Mix sequentializations in linear time. Nevertheless, by applying our bridge between proof nets and graph theory, we get the first *quasi-linear* time algorithm for MIXSEQ. The beginning of the next section will discuss why the problem seems harder with Mix.

Our algorithm proceeds in a "top-down" way: it starts by determining the root of the derivation tree and the link it introduces. To obtain the children of the root, it suffices to recurse on the connected components created by removing this link.

Furthermore, through the correspondence of Theorem 3.4, finding a link which is introduced by the last rule of some sequentialization amounts to finding a bridge in the matching of the graphification of the proof net (cf. §3.1). This is in fact a bit more convenient with graphifications than with general unique perfect matchings, thanks to the following property:

▶ **Lemma 5.1.** *All bridges in the graphification of some proof structure are matching edges.*

The algorithm will alternate between finding and deleting bridges; a deletion may cut cycles and thus create new bridges, which we want to detect without traversing the entire graph each time. To do so, we use a *dynamic bridge-finding data structure* designed for this kind of use case by Holm et al. [16]. It keeps an internal state corresponding to a graph, whose set of n vertices is immutable but whose set of edges may vary, and supports the following operations in $O((\log n)^2 (\log \log n)^2)$ amortized time:

- updating the graph by inserting or deleting an edge;
- computing the number of vertices of the connected component of a given vertex;
- finding a bridge in the connected component of a given vertex;
- determining whether two vertices are in the same connected component.

▶ **Theorem 5.2.** MIXSEQ *can be solved in* $O(n(\log n)^2 (\log \log n)^2)$ *time.*

Proof. Let π be a MLL+Mix proof net with n links, and $(G = (V, E), M)$ be its graphification. Both V and E are have cardinality $O(n)$ (in fact, $|V| = 2n$ and $|M| = n$).

The algorithm starts by initializing the bridge-finding data structure D with the graph G, computing the weakly connected components of π in linear time, and selecting a link in each

---

[9] It can find a feasible cycle, witnessing incorrectness, but cannot produce a certificate of correctness.

component. On each selected link l, we call the following recursive procedure; its role is to sequentialize the sub-proof net of π containing l whose graphification is a current connected component of G (G and D being mutable global variables):

- Let u be one endpoint of the matching edge corresponding to l. Using the bridge-finding structure, find a bridge $e = (v, w)$ in the component of u; necessarily, $e \in M$. Remove the edge e from G (and reflect this change on D with a deletion operation).
- If both v and w are isolated vertices, e corresponds to an `ax`-link and the entire sub-proof net consisted of this link. In this case, return a sequentialization with a single `ax`-rule.
- If one of v and w is isolated, and the other is not – by symmetry, let us assume the latter is v – then e corresponds to a $\otimes$-link l'. Let p and p' be its predecessors.
    - Remove all edges incident to v.
    - If the matching edges corresponding to p and p' are in the same connected component of G, recurse on p, add a final $\otimes$-link and return the resulting sequentialization.
    - If p and p' are in different connected components of G, recurse on p and p', use the results as the two premises of a Mix rule, add a final $\otimes$-link and return the resulting sequentialization.
- If neither v nor w is isolated, e corresponds to a $\otimes$-link. This is handled similarly to the $\otimes$+Mix case above.

Let us evaluate the time complexity. At each recursive call, one bridge is eliminated from G, so the number of recursive calls is n. The cost of each recursive call is $O(1)$ except for the updates and queries of the bridge-finding data structure. In total, there are $|E| = O(n)$ deletions, $|M| = n$ bridge queries, and at most n connectedness tests, and each of those takes $O((\log n)^2 (\log \log n)^2)$ amortized time. Hence the $O(n(\log n)^2(\log \log n)^2)$ bound. ◄

▶ **Remark.** If we want to compute a sequentialization for a unique perfect matching, in general, a complication is the existence of bridges which are not in the matching.

Interestingly, one can determine whether a bridge e is in M *without looking at M*: it is the case if and only if both of the connected components created by removing e have an odd number of vertices. This leads to an algorithm for UniquePM; it is virtually the same as the one proposed by Gabow et al. [11, §2][10], from which we took our inspiration.

▶ **Remark.** One needs to use a sparse representation for derivation trees: the size of a fully written-out sequent calculus proof is, in general, not linear in the size of its proof net.

6   On the kingdom ordering of links

One may wonder if we could not have just tweaked an algorithm for MLL sequentialization into an algorithm for MixSeq. In order to argue to the contrary, let us briefly mention a difference between Bellin and van de Wiele's study of the sub-proof nets of MLL proof nets [5] and its extension to the MLL+Mix case by Bellin [4]. Any MLL sub-proof net of a MLL proof net may appear in the sequentialization of the latter; however, for MLL+Mix, Figure 7 serves as a counterexample: the sub-proof structure containing all links but the $\otimes$-link is correct for MLL+Mix, but it cannot be an intermediate step in a sequentialization of the entire proof net. A *normality* condition is needed to distinguish those sub-proof nets

---

[10] Not to be confused with their algorithm for UniquenessPM [11, §3] that we used in §4.1. They only claim a bound of $O(m \log^4 n)$ because the best dynamic 2-edge-connectivity data structure known at the time has operations in $O(\log^4 n)$ amortized time.
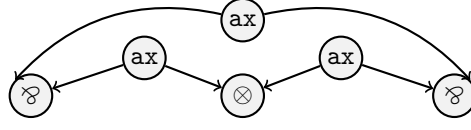
Figure 7 A MLL+Mix proof net which highlights a difficulty in solving MixSeq.

which may appear in a sequentialization, and this is why sequentialization algorithms which are morally based on a greedy parsing strategy, such as Guerrini's linear-time algorithm [14], do not adapt well to the presence of the Mix rule.

Any link l in a MLL+Mix proof net π admits a minimum normal sub-proof net of π containing l, its *kingdom* [4]. Bellin's *kingdom ordering* is the partial order on links corresponding to the inclusion between kingdoms. We give an algorithm to compute this order for any MLL+Mix proof net: this is yet another application of matching theory. It uses a characterization of the kingdom ordering in terms of a relation called *dependency* by Bagnol et al. [1] (who, in turn, take this name from the closely related *dependency graph* of Mogbil and Naurois [17]). We will also see how this dependency relation can be reformulated, through our correspondence between proof structures and perfect matchings, in terms of the *blossoms* mentioned in §2.1 and §4.1.

One may in fact define the kingdom ordering, written $\ll_\pi$, without reference to the notion of normal sub-proof net (we will not introduce the latter formally here):

▶ **Definition 6.1.** Let π be a MLL+Mix proof net. For any two links p, q of π, $p \ll_\pi q$ if and only if, in any sequentialization of π, the rule introducing q has, among its premises, a proof net containing p.

From this point of view, the kingdom ordering gives us information about the set of all sequentializations. Let us give some examples. The proof net of Figure 4b admits a unique sequentialization, so this directly gives us the kingdom ordering: for instance the middle $\otimes$-link is the greatest element. On the other hand, in the proof net of Figure 7, both $\invamp$-links may be introduced by a last rule, so there is no greatest element. In fact, the kingdom ordering coincides with the predecessor relation. So it does not distinguish between the 3 terminal links even though, unlike the 2 others, the $\otimes$-link cannot be introduced last.

Before proceeding further, here is another property of MLL proof nets which is contradicted by Figure 7 for MLL+Mix proof nets, providing more evidence that MixSeq is trickier than MLL sequentialization.

▶ **Proposition 6.2.** *Let π be a MLL proof net and l be a maximal link for $\ll_\pi$. Then there exists a sequentialization of π whose last rule introduces l.*

6.1 Computing the kingdom ordering

▶ **Definition 6.3.** Let π be a proof structure. We write $D(\pi)$ for the *dependency relation* defined as follows: for any two links $p \neq q$ of π, p *is a dependency of* q when q is a $\invamp$-link and there exists a feasible path between the predecessors of q going through p.

For instance, in the proof net of Figure 4b, the left $\invamp$-link depends on the left $\otimes$-link, but not on the other $\otimes$-links or $\invamp$-links; the middle $\otimes$-link has no dependency. In the case of Figure 7, the dependency relation is empty.

▶ **Theorem 6.4** (Bellin [4, Lemma 2][11]). *Let π be a* MLL+Mix *proof net. The transitive closure of $D(\pi) \cup S(\pi)$ is $\ll_\pi$, where $(p, q) \in S(\pi)$ means that p is a predecessor of q.*

The dependency relation can be computed by reduction to a matching problem *in the case of MLL+Mix proof nets*: even though it is well-defined in arbitrary proof structures, we need MLL+Mix correctness to compute it, because our matching algorithm relies on the absence of alternating cycles.

▶ **Lemma 6.5.** *Let G be a graph with a* unique *perfect matching M, and $e, f, g \in M$ be pairwise distinct edges. The existence of an alternating path starting with e, ending with f and crossing g can be reduced to the existence of a perfect matching.*

Proof. Let $(u, v) = e$, $(u', v') = f$ and $(a, b) = g$. Let G' be the graph obtained by adding new vertices s and s', new edges (s, u), (s, v), (s', u') and (s', v'), and by removing g. This graph admits a matching $M' = M \setminus \{g\}$ leaving the 4 vertices s, s', a and b unmatched.

Suppose G' admits a perfect matching M''. Then the symmetric difference $M' \triangle M''$ consists of two vertex-disjoint alternating paths for M', whose endpoints are $\{s, s', a, b\}$ by the same reasoning as Lemma 2.2. (In general, the symmetric difference may also contain alternating cycles, but if it were the case here, M would not be unique.)

We claim that these paths either go from s to a and b to s', or from s to b and a to s'. Otherwise, there would be an alternating path from a to b for M', and together with the matching edge g we removed earlier, this would give us an alternating cycle for M in G.

In both cases, let us join the two paths together by adding g, and remove the edges incident to s and s'. We get a path starting with e, ending with f, crossing g and alternating for M in G. Conversely, from such a path, one can get a perfect matching in G'.   ◀

▶ **Theorem 6.6.** *Let π be a* MLL+Mix *proof net with a link p and a $\bindnasrepma$-link q. Deciding whether $(p, q) \in D(\pi)$ can be done in linear time, in randomized* NC *and in* quasi-NC.

Proof. A degenerate case is when p is a predecessor of q: in this case, p depending on q is equivalent to π becoming incorrect if q is turned into a $\otimes$-link, and thus the complexity is the same as that of (the complement of) the correctness problem.

When p is not a predecessor of q, the definition of dependency translates into the problem defined in the above lemma by taking the graphification of π. Since the existence of a perfect matching can be decided in randomized NC or quasi-NC (cf. §2.1), so can our problem. To get a linear time complexity, we exploit the fact that we know a matching of G' leaving $O(1)$ vertices unmatched: a perfect matching can then be found with $O(1)$ iterations of an algorithm using *augmenting paths*, each iteration taking linear time, see e.g. [31, Chapter 9].   ◀

A transitive closure can be computed in polynomial time, and reachability in a directed graph can be decided in NL $\subset$ quasi-NC, so we get in the end:

▶ **Corollary 6.7.** *There are a polynomial-time algorithm and a* quasi-NC *algorithm to compute the kingdom ordering $\ll_\pi$ of any MLL+Mix proof net π.*

---

[11] This theorem was rediscovered in the special case of MLL proof nets by Bagnol et al. [1, Theorem 11], who refer to the kingdom ordering as the "order of introduction". We borrow the notations $D(\pi)$ and $S(\pi)$ from them.

6.2 Dependencies and blossoms in unique perfect matchings

We will now see how, through the correspondence of §3, Bellin's theorem can be rephrased as a statement on unique perfect matchings.

▶ **Definition 6.8.** Let G be a graph and M be a perfect matching of G. A *blossom* for M is a cycle whose vertices are all matched within the cycle, except for one, its *root*. The matching edge incident to the root, is called the *stem* of the blossom.

That is, a blossom consists of an alternating path between two vertices, starting and ending with a matching edge, together with a non-matching edge from the root to each of these two vertices; for instance, in Figure 1b, the two triangles are blossoms with a common stem. The stem of a blossom is not part of the cycle. Blossom are central to combinatorial matching algorithms, e.g. [9, 11], as we have previously mentioned.

▶ **Definition 6.9.** When $e \in M$ is in some blossom with stem $f \in M$, we write $e \to f$.

This is the graph-theoretical counterpart of the dependency relation, as is shown by the following two propositions.

▶ **Proposition 6.10.** *Let π be a MLL+Mix proof net and (G, M) be its graphification. Let p, q be links in π with corresponding matching edges $e_p, e_q \in M$. Then $e_p \to e_q$ if and only if p is a dependency of q or a predecessor of q, i.e. $(p, q) \in D(\pi) \cup S(\pi)$.*

▶ **Proposition 6.11.** *Let G be a graph, M be a perfect matching of G and π be the proofification of (G, M). Let $e, f \in M$ with corresponding $\otimes$-links $l_e, l_f \in M$. Then $e \to f$ if and only if l_e is a dependency of some $\otimes$-link q from which l_f is reachable (by a directed path).*

▶ **Remark.** In Proposition 6.10, the "if" direction holds even for incorrect proof structures; in Proposition 6.11, note that no uniqueness property is required of the perfect matching.

Thus, we see that Bellin's theorem is equivalent to the following theorem where $\to^+$ is the transitive closure of $\to$. As far as we know, this is a new result in graph theory.

▶ **Theorem 6.12.** *Let G be a graph with a* unique *perfect matching M, and $e, f \in M$. The edge e occurs before f in all sequentializations for M if and only if $e \to^+ f$.*

We can also formulate the theorem without mentioning sequentializations. Fix an edge $e \in M$, and iteratively remove the endpoints of any bridge in M, except e, until we end up with a vertex-induced subgraph of G where no bridge remains except e. This is the graph-theoretic analogue of the *kingdom* of e. Bellin's theorem says that for any edge f in the kingdom of e, $f \to^+ e$, that is:

▶ **Theorem 6.13.** *Let G be a graph with a* unique *perfect matching M. Suppose that M contains only one bridge e. Then for all $f \in M \setminus \{e\}$, $f \to^+ e$.*

This is the case in Figure 1b: the middle edge e is the only bridge, and it is the stem of the two triangular blossoms which contain the other matching edges.

The graph-theoretic versions are somewhat simpler to state than the original theorem: one takes the transitive closure of a single relation, instead of a union of two unrelated relations. We give a direct proof of the last formulation in Appendix D, based on the *blossom shrinking* operation mentioned in §4.1.

7 Conclusion

We have presented a correspondence between proof nets and perfect matchings, and demonstrated its usefulness through several applications of graph theory to linear logic: our results give the best known complexity for MLL+Mix correctness and sequentialization, by taking advantage of sophisticated graph algorithms. We also expect linear logic to eventually lead to new results in graph theory through this correspondence – in fact, the rephrasing of Bellin's theorem in the last section is one such example – and in general, we hope to see fruitful interactions arise between those two domains.

Perspectives. Now that we have shed a new light on MLL+Mix proof nets, it would be interesting to revisit the well-studied theory of MLL proof nets. Therefore, we would like to find the right graph-theoretical counterpart to the connectedness condition in the Danos–Regnier criterion for MLL, but unique perfect matchings do not seem to be the right setting to do so. Indeed, there are many objects in graph theory which admit "structure from acyclicity" theorems equivalent to Kotzig's theorem on unique perfect matchings (cf. [30]) – that is, to MLL+Mix sequentialization – and some of these may be better suited to go beyond MLL+Mix proof nets and extend the correspondence, either to MLL or to larger fragments of linear logic.

In particular, we have already taken inspiration from *edge-colored graphs* (see e.g. [2, §16]), which are rather close to the usual paired graphs, to prove the coNP-hardness of Pagani's *visible acyclicity* condition [24] on MELL proof structures (cf. the workshop abstract [23]). Let us also mention that we have found an interpretation of Retoré's construction [27] in terms of graphs with *forbidden transitions* [29], which can be seen as the generalization of paired graphs by dropping the disjointness condition.

A closely related question is that of finding a natural graph-theoretic decision problem equivalent to correctness for MLL without Mix through low-complexity reductions – hopefully computable both in linear time and in AC^0, like the equivalence between MixSeq and UniquenessPM exhibited in this paper. Though both Murawski & Ong [22] and Mogbil & Naurois [17] reduce MLL correctness to problems on directed graphs, the complexity of these reductions is higher than we would like: the first is not computed in logarithmic space, the second uses a subroutine for undirected connectivity, a L-complete problem whose membership in L is highly non-trivial. An answer to this question may help clarify why all known linear-time correctness criteria for MLL rely on the same sophisiticated data structure, as mentioned in §4.1.

References

1   Marc Bagnol, Amina Doumane, and Alexis Saurin. On the dependencies of logical rules. In *FOSSACS, 18th International Conference on Foundations of Software Science and Computation Structures*, 2015.

2   Jørgen Bang-Jensen and Gregory Gutin. *Digraphs. Theory, algorithms and applications. 2nd ed.* London: Springer, 2nd ed. edition, 2009.

3   David A. Mix Barrington. Quasipolynomial size circuit classes. In *[1992] Proceedings of the Seventh Annual Structure in Complexity Theory Conference*, pages 86–93, 1992.

4   Gianluigi Bellin. Subnets of proof-nets in multiplicative linear logic with MIX. *Mathematical Structures in Computer Science*, 7(6):663–669, 1997.

5 Gianluigi Bellin and Jacques van de Wiele. Subnets of Proof-nets in MLL-. In *Proceedings of the Workshop on Advances in Linear Logic*, pages 249–270, New York, NY, USA, 1995. Cambridge University Press.

6 Ashok K. Chandra, Larry Stockmeyer, and Uzi Vishkin. Constant depth reducibility. *SIAM Journal on Computing*, 13(2):423–439, 1984.

7 Vincent Danos. *La Logique Linéaire appliquée à l'étude de divers processus de normalisation (principalement du Lambda-calcul)*. PhD thesis, Université Paris-Diderot – Paris VII, 1990.

8 Vincent Danos and Laurent Regnier. The structure of multiplicatives. *Archive for Mathematical Logic*, 28(3):181–203, 1989.

9 Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17(0):449–467, 1965.

10 Arnaud Fleury and Christian Retoré. The mix rule. *Mathematical Structures in Computer Science*, 4(2):273–285, 1994.

11 Harold N. Gabow, Haim Kaplan, and Robert E. Tarjan. Unique maximum matching algorithms. *Journal of Algorithms*, 40(2):159–183, 2001.

12 Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, apr 1985.

13 Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, jan 1987.

14 Stefano Guerrini. A linear algorithm for MLL proof net correctness and sequentialization. *Theoretical Computer Science*, 412(20):1958–1978, apr 2011.

15 Thanh Minh Hoang, Meena Mahajan, and Thomas Thierauf. On the bipartite unique perfect matching problem. In *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part I*, pages 453–464, 2006.

16 Jacob Holm, Eva Rotenberg, and Mikkel Thorup. Dynamic bridge-finding in $\tilde{O}(\log^2 n)$ amortized time. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, Proceedings, pages 35–52. Society for Industrial and Applied Mathematics, jan 2018.

17 Paulin Jacobé de Naurois and Virgile Mogbil. Correctness of linear logic proof structures is NL-complete. *Theoretical Computer Science*, 412(20):1941–1957, apr 2011.

18 Anton Kotzig. Z teórie konečných grafov s lineárnym faktorom. II. *Matematicko-fyzikálny časopis*, 09(3):136–159, 1959.

19 Dexter Kozen, Umesh V. Vazirani, and Vijay V. Vazirani. NC algorithms for comparability graphs, interval graphs, and testing for unique perfect matching. In *Foundations of Software Technology and Theoretical Computer Science, Fifth Conference, New Delhi, India, December 16-18, 1985, Proceedings*, pages 496–503, 1985.

20 László Lovász. On determinants, matchings, and random algorithms. In *Fundamentals of Computation Theory*, pages 565–574, 1979.

21 Ketan Mulmuley, Umesh V. Vazirani, and Vijay V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–113, 1987.

22 Andrzej S. Murawski and C.-H. Luke Ong. Fast verification of MLL proof nets via IMLL. *ACM Transactions on Computational Logic*, 7(3):473–498, 2006.

23 Lê Thành Dũng Nguyễn. On the complexity of finding cycles in proof nets. Extended abstract for the workshop Developments in Implicit Computational Complexity 2018.

24 Michele Pagani. Visible acyclic differential nets, Part I: Semantics. *Annals of Pure and Applied Logic*, 163(3):238–265, 2012.

25 Michael O. Rabin and Vijay V. Vazirani. Maximum matchings in general graphs through randomization. *Journal of Algorithms*, 10(4):557–567, dec 1989.

26 Christian Retoré. Handsome proof-nets: R&B-graphs, perfect matchings and series-parallel graphs. Research Report, INRIA, 1999.

27   Christian Retoré. Handsome proof-nets: perfect matchings and cographs. *Theoretical Computer Science*, 294(3):473–488, 2003.

28   Ola Svensson and Jakub Tarnawski. The matching problem in general graphs is in quasi-NC. In Chris Umans, editor, *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pages 696–707. IEEE Computer Society, 2017.

29   Stefan Szeider. Finding paths in graphs avoiding forbidden transitions. *Discrete Applied Mathematics*, 126(2-3):261–273, 2003.

30   Stefan Szeider. On theorems equivalent with Kotzig's result on graphs with unique 1-factors. *Ars Combinatoria*, 73:53–64, 2004.

31   Robert Endre Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.

A   Proofs of §3

Proof of Proposition 3.2. By negating the two sides of the equivalence, the goal becomes proving that a proof structure π contains a feasible cycle if and only if its graphification (G, M) contains an alternating cycle.

Consider any alternating cycle for M in G of length $2n$, and take the $\mathbb{Z}/(n)$-indexed sequence of vertices corresponding to the matching edges in the cycle. By construction of the graphification, if two edges in M are incident to a common non-matching edge, then the corresponding links in π are adjacent: thus, in our sequence, each vertex is adjacent to the previous and the next one, and thus we have a cycle. If it were not feasible, it would contain three consecutive links p, q, r with q a $⅋$-link and p, r its predecessors[12]; but then the alternating cycle would have to cross two incident non-matching edges (from p to q and from q to r), which is impossible. Thus, π contains a feasible cycle.

To show the converse we will exhibit a right inverse to the map from alternating cycles to feasible cycles defined above. Consider a feasible cycle: it can be partitioned into directed paths from `ax`-links to $\otimes$-links. Let l be an intermediate link in such a path, and e, p, s be matching edges corresponding respectively to l, its predecessor, and its successor in the directed path. s has a unique endpoint u which is incident to both endpoints of e; e has a unique endpoint v which is *not* incident to both endpoints of p. To join e with s, we use the edge (u, v). By taking all these non-matching edges for all maximal directed paths in the cycle, as well as a choice of two edges incident to each matching edge corresponding to an `ax`-link, and the matching edges (a_l, b_l) corresponding to all the links l in the cycle, we get an alternating cycle.   ◀

Proof of Lemma 3.3. Suppose for contradiction that l is not a terminal link, and let l' be a successor of l. Then for some endpoint v of $(a_{l'}, b_{l'})$, (a_l, v) and (b_l, v) are both edges in G, and they make up a path between a_l and b_l not going through (a_l, b_l). Thus, (a_l, b_l) cannot be a bridge.

The fact that (a_l, b_l) is a bridge means that by removing this edge, a_l and b_l are in different connected components; if l is a $\otimes$-link, each of these connected components contain the matching edge corresponding to one premise of l.   ◀

---

[12] To expand on this point: this is because we have prohibited vertex repetitions in our definition of cycles. This is legitimate since a graph is a forest if and only if it does not contain a non-vertex-repeating cycle.

Proof of Theorem 3.4. We convert a sequentialization S of (G, M) into a sequentialization Σ of π inductively as follows. Since $G \neq \emptyset$, the last rule of S is either a disjoint union or the introduction of a bridge $e = (a_l, b_l) \in M$ by joining together (G_a, M_a) and (G_b, M_b) with respective sequentializations S_a and S_b. In the latter case, l is a terminal link of π.

- If $G_a = G_b = \emptyset$, then l is an $\mathtt{ax}$-link, and Σ consists of a single $\mathtt{ax}$-rule.
- If $G_a \neq \emptyset$ and $G_b = \emptyset$, then l is a $\mathbin{\rotatebox[origin=c]{180}{\&}}$-link, and the removal of l from π yields a proof structure π' whose graphification is (G_a, M_a). Σ then consists of a $\mathbin{\rotatebox[origin=c]{180}{\&}}$-rule introducing l applied to the sequentialization of π' corresponding to S_a.
- If $G_a \neq \emptyset$ and $G_b \neq \emptyset$, then l is a $\otimes$-link. Since e is a bridge, the removal of l from π yields two proof structures π_a and π_b whose respective graphifications are (G_a, M_a) and (G_b, M_b). Σ then consists of an $\otimes$-rule applied to the translations of S_a and S_b.

If the last rule of S is a disjoint union rule, it is translated into a Mix rule in Σ.

The bijectivity can be proven by defining the inverse transformation and by checking that it is indeed its inverse. ◀

Proof of Proposition 3.6. Let π be the proofification of (G, M). Any feasible cycle in π changes direction only at $\mathtt{ax}$-links and $\otimes$-links, and therefore can be partitioned into an alternation of $\otimes$-links, corresponding to matching edges, and of paths starting with some B_u, ending with some B_v and crossing some $\mathtt{ax}_e$, corresponding to non-matching edges $e = (u, v)$. Therefore, it corresponds to an alternating cycle for M, and the mapping defined this way is bijective. ◀

Proof of Proposition 3.7. This follows from the fact that a $\otimes$-link may be introduced by the last rule of a sequentialization if and only if it is splitting, i.e. its removal disconnects its two precedessors. ◀

B   Omitted proof in §5

Proof of Lemma 5.1. Let e be a non-matching edge. Then there are matching edges (u, v) and (s, t) such that the link corresponding to (u, v) is the predecessor of the one for (s, t), and $e = (u, s)$. The non-matching edge (v, s) is then also present in the graph, and so e cannot be a bridge. ◀

C   Omitted proofs in §6

Proof of Proposition 6.2. If l is a terminal $\mathbin{\rotatebox[origin=c]{180}{\&}}$-link, no other assumption is needed for the existence of such a sequentialization. Else, l is a terminal $\otimes$-link and it suffices to show that l is *splitting*, i.e. that the removal of l splits π into two connected components.

Suppose that it is not the case, and consider some sequentialization of π: it must contain a $\mathbin{\rotatebox[origin=c]{180}{\&}}$-rule, applied to a sub-proof net π' for which l is splitting, which turns it into a sub-proof net for which l is not splitting anymore. Let p be the $\mathbin{\rotatebox[origin=c]{180}{\&}}$-link introduced by that rule; its predecessors lie in different connected components of $\pi' \setminus \{l\}$. Since π' is a MLL proof net, the predecessors of p are connected by a feasible path in π', which must cross l. This shows that l is a dependency of p in the sense of Definition 6.3, contradicting the maximality of l. (This only uses the fact that $D(\pi) \subseteq \ll_\pi$, which is the "easy" part of Bellin's theorem.) ◀

Proof of Proposition 6.10. If $(p, q) \in S(\pi)$, then by construction there exists a blossom of length 3 containing p with stem q. If $(p, q) \in D(\pi)$, then for the same reason as Proposition 3.2, we can get, from the feasible path between the predecessors of q visiting p, an alternating path for M starting and ending with the edges corresponding to those predecessors and crossing

the edge corresponding to p. By adding two non-matching edges to the same endpoint of the matching edge for q, we get a blossom with stem q.

Conversely, let q be a link, e the corresponding matching edge, and B be a blossom with stem q. Let us first note that if B contains an non-matching edge joining e with the matching edge corresponding to a successor of q, then by replacing this non-matching edge with its twin incident to the other endpoint of q, we get an alternating cycle; this is impossible because we have assumed π to be a MLL+Mix proof net. Therefore, the first and last matching edges in B are both premises of q. If they are the same – that is, if B has length 3 and contains a single matching edge – then this edge corresponds to a predecessor p of q. Otherwise, B gives an alternating path between two distinct premises of q; necessarily q is a $\invamp$-link (otherwise, there would be an alternating cycle), and all links corresponding to matching edges in B are dependencies of q.   ◀

Proof of Proposition 6.11. Let B be a blossom with stem f, whose two non-matching edges incident to f are a and b. B translates into a feasible path between $\mathtt{ax}_a$ and $\mathtt{ax}_b$ in π. Now, $\mathtt{ax}_a$ and $\mathtt{ax}_b$ are also leaves of a binary tree of $\invamp$-links whose root has the single successor l_f; by taking q to be the lowest common ancestor of $\mathtt{ax}_a$ and $\mathtt{ax}_b$ in this tree, l_f is reachable from q, and every link in the path between $\mathtt{ax}_a$ and $\mathtt{ax}_b$ depends on q. Conversely, any feasible path between the two premises of a $\invamp$-link corresponds to a blossom for M in G.   ◀

D   Proof of the graph-theoretic Bellin theorem (Theorem 6.13)

Let G be a graph with a unique perfect matching M, containing a single bridge e. Removing the edge e, *but not its endpoints*, results in two connected components which both have a unique *near-perfect matching* (leaving one vertex unmatched) containing no bridge. If both these components have a single vertex, then the theorem is vacuously true; else, we have reduced it to the following proposition, where $a \leftarrow b$ means that $b \to a$, $\leftarrow^*$ is the reflexive transitive closure of $\leftarrow$, and $u \leftarrow f$ means that f is contained in a blossom with root u.

▶ **Proposition.** *Let G be a graph with a near-perfect matching M and let u be the unmatched vertex. Suppose G has no bridge in M and no alternating cycle for M. Then for all $f \in M$, there exists $g \in M$ such that $u \leftarrow g \leftarrow^* f$.*

The proof of this proposition relies on the *blossom shrinking* operation: starting from the graph G with a matching M, this consists in taking the quotient graph G' where all the vertices of the blossom have been identified; M induces a matching M' in G'.

▶ **Lemma.** *Under the hypotheses of the proposition, if $M \neq \emptyset$, then:*
1. *There exists a blossom in G for M with root u.*
2. *Let G' be the graph obtained by shrinking this blossom, with induced matching $M' \subset M$. There is no bridge in M' and no alternating cycle for M'.*
3. *Let u' be the exposed vertex in G', corresponding to the shrunk blossom. For all $f \in M'$ with $u' \leftarrow f$ in G', there exists $g \in M$ such that $u \leftarrow g \leftarrow^* f$ in G.*

Proof of (1). The absence of alternating cycle amounts to saying that M is the unique perfect matching of $G[V \setminus \{u\}]$ where V is the vertex set of G. (Note that $M \neq \emptyset \Leftrightarrow V \setminus \{u\} \neq \emptyset$.) By Kotzig's theorem, M contains a bridge e of $G[V \setminus \{u\}]$; let V_1 and V_2 be the connected components created by the removal of e (but keeping its endpoints) from $G[V \setminus \{u\}]$. We create a new graph H by starting from $G[V \setminus \{u\}]$, adding two new vertices u_1 and u_2, and adding the edges (u_i, v) for all $v \in V_i$ with v adjacent to u in G ($i = 1, 2$), and the edge (u_1, u_2).

The perfect matching $M \cup \{(u_1, u_2)\}$ of H contains no bridge of H: since e is not a bridge of G, there is at least one edge between u_1 and V_1 and one edge between u_2 and V_2, so (u_1, u_2) is not a bridge in H; and any edge of M would be a bridge of G if it were a bridge of H. Let us apply Kotzig's theorem again: this perfect matching admits an alternating cycle, which cannot be contained in $H[V \setminus \{u\}] = G[V \setminus \{u\}]$. Therefore, it contains an alternating path from u_1 to u_2, from which we retrieve a blossom with root u in G.   ◀

Proof of (2). If there existed a bridge $e \in M'$ of G', then $G' \setminus \{e\}$ would be disconnected while $G \setminus \{e\}$ would be connected; this is impossible. An alternating cycle for M' would not visit u' because it is unmatched, and therefore would be an alternating cycle for M in G.   ◀

Proof of (3). Let B be the blossom with root u in G that has been shrunk, and B' be the blossom with root u' in G' containing f. There are two non-matching edges e'_1 and e'_2 in B' incident to u'; let $e_1 = (u_1, v_1)$ be a preimage of e'_1 and $e_2 = (u_2, v_2)$ be a preimage of e'_2 in G, with $u_1, u_2 \in B$.

The blossom B can be decomposed into $P_1 \cup Q \cup P_2$, where P_1 is an alternating path from u to u_1 (possibly empty, if $u = u_1$), Q is an alternating path from u_1 to u_2 (possibly empty, if $u_1 = u_2$), and P_2 is an alternating path from u_2 to u. As for B', it lifts to an alternating path R between u_1 and u_2 starting and ending with a non-matching edge, so that $|R|$ is odd and $f \in R$. We proceed by case analysis on the parity of $|P_1|$ and $|P_2|$.

- If they are both even, then $P_1 \cup R \cup P_2$ is a blossom: $u \twoheadleftarrow f \leftarrow^* f$.
- If $|P_1|$ is even and $|P_2|$ is odd, then $Q \cup R$ is a blossom with root u_1. Either $u_1 = u$ and then $u \twoheadleftarrow f$, or there is an edge $g \in B \cap M$ incident to u_1 and then $u \twoheadleftarrow g \leftarrow f$.
- The case $|P_1|$ even and $|P_2|$ odd is symmetric to the previous one.
- If they were both odd, $Q \cup R$ would be an alternating cycle.   ◀

Proof of the proposition. By induction on the size of G.

Let us take a blossom using lemma (1). If it contains f, then $u \twoheadleftarrow f$ and we are done. Else, we shrink the blossom and get G', M' and u'; by lemma (2), they satisfy the assumptions of the proposition. By the induction hypothesis, there exists g such that $u' \twoheadleftarrow g \leftarrow^* f$ in G'. Thanks to lemma (3), $u' \twoheadleftarrow g$ entails $u \twoheadleftarrow h \leftarrow^* g$ in G for some $h \in M$. Also, $g \leftarrow^* f$ in G' entails $g \leftarrow^* f$ in G because the (possibly empty) sequence of blossoms which binds f to g in G' cannot contain the vertex u', and therefore lifts to exactly the same edges in G. Thus, $u \twoheadleftarrow h \leftarrow^* g \leftarrow^* f$ and therefore $u \twoheadleftarrow h \leftarrow^* f$ in G.   ◀

Narrowing Trees for Syntactically Deterministic Conditional Term Rewriting Systems

Naoki Nishida

Graduate School of Informatics, Nagoya University, Nagoya, Japan
nishida@i.nagoya-u.ac.jp
https://orcid.org/0000-0001-8697-4970

Yuya Maeda

Graduate School of Informatics, Nagoya University, Nagoya, Japan
yuya@trs.css.i.nagoya-u.ac.jp

──── Abstract ────

A narrowing tree for a constructor term rewriting system and a pair of terms is a finite representation for the space of all possible innermost-narrowing derivations that start with the pair and end with non-narrowable terms. Narrowing trees have grammar representations that can be considered regular tree grammars. Innermost narrowing is a counterpart of constructor-based rewriting, and thus, narrowing trees can be used in analyzing constructor-based rewriting to normal forms. In this paper, using grammar representations, we extend narrowing trees to syntactically deterministic conditional term rewriting systems that are constructor systems. We show that narrowing trees are useful to prove two properties of a normal conditional term rewriting system: one is infeasibility of conditional critical pairs and the other is quasi-reducibility.

1 Introduction

Conditional term rewriting [32, Chapter 7] is known to be more complicated than unconditional term rewriting in the sense of analyzing properties, e.g., *operational termination* [21] (*quasi-decreasingness* [32]), *confluence* [37], and *reachability* [6]. A popular approach to the analysis of conditional term rewriting systems (CTRSs, for short) is to transform a CTRS into an unconditional term rewriting system (a TRS, for short) that is in general an overapproximation of the CTRS w.r.t. reduction (cf. [32]). This approach enables us to use existing techniques for the analysis of TRSs. For example, a CTRS is operationally terminating if the *unraveled* TRS [22, 32] is terminating [5]. To prove termination of the unraveled TRS, we can use many techniques for proving termination of TRSs (cf. [32]). On the other hand, it is not so easy to analyze reachability which is relevant to, e.g., *infeasibility* of conditions – non-existence of substitutions satisfying conditions – of conditional rewrite rules, conditional critical pairs, etc.

Let us consider to prove confluence of the following *normal* 1-CTRS [31] defining *even* and *odd* predicates over the non-negative integers represented by 0 and s:

$$\mathcal{R}_1 = \left\{ \begin{array}{llll} e(0) \to \text{true}, & e(s(x)) \to \text{true} \Leftarrow o(x) \twoheadrightarrow \text{true}, & e(s(x)) \to \text{false} \Leftarrow e(x) \twoheadrightarrow \text{true}, \\ o(0) \to \text{false}, & o(s(x)) \to \text{true} \Leftarrow e(x) \twoheadrightarrow \text{true}, & o(s(x)) \to \text{false} \Leftarrow o(x) \twoheadrightarrow \text{true} \end{array} \right\}$$

Unfortunately, neither a transformational approach in [10, 9] nor a direct approach to reachability analysis to prove infeasibility of conditional critical pairs succeeds in proving confluence of $\mathcal{R}_1$. For example, $\mathcal{R}_1$ has the following four critical pairs:

$$\langle \text{true}, \text{false} \rangle \Leftarrow o(x) \twoheadrightarrow \text{true} \ \& \ e(x) \twoheadrightarrow \text{true} \qquad \langle \text{false}, \text{true} \rangle \Leftarrow o(x) \twoheadrightarrow \text{true} \ \& \ e(x) \twoheadrightarrow \text{true}$$
$$\langle \text{true}, \text{false} \rangle \Leftarrow e(x) \twoheadrightarrow \text{true} \ \& \ o(x) \twoheadrightarrow \text{true} \qquad \langle \text{false}, \text{true} \rangle \Leftarrow e(x) \twoheadrightarrow \text{true} \ \& \ o(x) \twoheadrightarrow \text{true}$$

An operationally terminating CTRS is confluent if all critical pairs of the CTRS are infeasible (cf. [1, 4]). To prove infeasibility of the critical pairs above, it suffices to show non-existence of terms t such that $o(t) \to^*_{\mathcal{R}_1} \text{true}$ and $e(t) \to^*_{\mathcal{R}_1} \text{true}$. Thanks to the meaning of *even* and *odd* predicates, it would be easy for human to notice that such a term t does not exist. However, it is not so easy to mechanize a way to show non-existence of t. In fact, confluence provers for CTRSs, ConCon [35], CO3 [25], and CoScart [8], based on e.g., transformations of CTRSs into TRSs and/or reachability analysis for infeasibility of conditional critical pairs failed to prove confluence of $\mathcal{R}_1$ (see Confluence Competition 2016 [1] and 2017,[2] 489.trs or 522.trs). Note that a *semantic approach* in [19, 18] can prove confluence of $\mathcal{R}_1$ using AGES [11], a tool for generating logical models of order-sorted first-order theories (cf. [20]).

The (non-)existence of a term t with $o(t) \to^*_{\mathcal{R}_1} \text{true}$ and $e(t) \to^*_{\mathcal{R}_1} \text{true}$ can be reduced to the (non-)existence of substitutions θ such that $o(x) \rightsquigarrow^*_{\theta, \mathcal{R}_1} \text{true}$ and $e(x) \rightsquigarrow^*_{\theta, \mathcal{R}_1} \text{true}$, where $\rightsquigarrow$ denotes the *narrowing* step [14]. In addition, the non-existence of such substitutions can be reduced to the emptiness of the set of such substitutions, i.e., the emptiness of $\{\theta \mid o(x) \rightsquigarrow^*_{\theta, \mathcal{R}_1} \text{true}, \ e(x) \rightsquigarrow^*_{\theta, \mathcal{R}_1} \text{true}\}$. From this viewpoint, the enumeration of substitutions obtained by narrowing from a pair of terms would be useful in analyzing rewriting sequences that starts with instances of the pair.

A *narrowing tree* [29] for a sufficiently complete constructor TRS $\mathcal{R}$ with the root pair $s \twoheadrightarrow t$ where t is a constructor term is a finite representation that defines the set of substitutions θ such that the pair $s \twoheadrightarrow t$ narrows to a special constant $\top$ by *innermost* narrowing $\overset{i}{\rightsquigarrow}_{\mathcal{R}}$ with a substitution θ (i.e., $(s \twoheadrightarrow t) \overset{i}{\rightsquigarrow}^*_{\theta, \mathcal{R}} \top$ and thus $\theta s \overset{c}{\to}^*_{\mathcal{R}} \theta t$). Note that $\twoheadrightarrow$ is considered a binary symbol, $(x \twoheadrightarrow x) \to \top$ is assumed to be implicitly included in $\mathcal{R}$, and $\overset{c}{\to}_{\mathcal{R}}$ denotes the *constructor-based rewriting* step which applies rewrite rules to *basic* terms. Note that a basic terms is of the form $f(u_1, \ldots, u_n)$ with a defined symbol f and constructor terms $u_1, \ldots, u_n$. A narrowing tree can be the enumeration of substitutions obtained by innermost narrowing of $\mathcal{R}$ to $\top$. The idea of narrowing trees has been extended to finite representations of SLD trees for logic programs [30].

In this paper, we extend narrowing trees to *syntactically deterministic conditional* term rewriting systems (a SDCTRS, for short) that are constructor systems. The class of SDCTRSs is reasonable to model functional programs. We do not directly extend narrowing trees to conditional systems, but we convert an SDCTRS to an equivalent unconditional constructor system that may have extra variables. Narrowing trees for the converted constructor system can be used for the original SDCTRS, i.e., they represent all substitutions derived by innermost narrowing of the original SDCTRS.

---

[1] http://cops.uibk.ac.at/results/?y=2016&c=CTRS
[2] http://cops.uibk.ac.at/results/?y=2017-full-run&c=CTRS
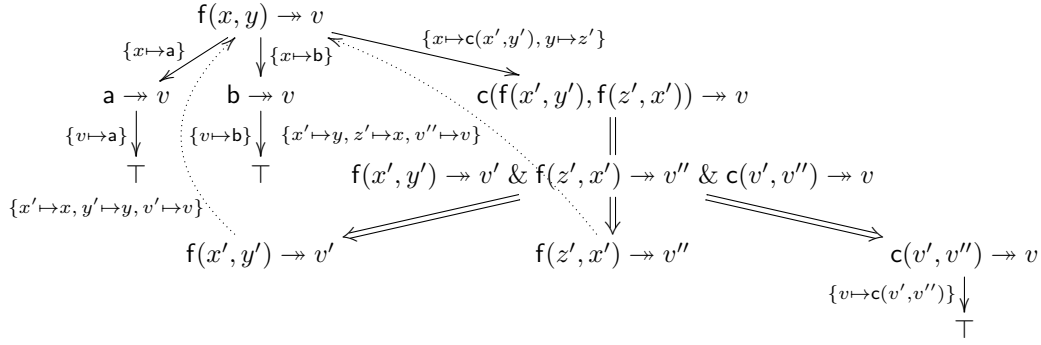
Figure 1 A narrowing tree for $\mathsf{f}(x,y) \twoheadrightarrow v$ w.r.t. $\mathcal{R}_2$.

Consider the sufficiently complete constructor TRS $\mathcal{R}_2 = \{ \mathsf{f}(\mathsf{a},z) \to \mathsf{a},\ \mathsf{f}(\mathsf{b},z) \to \mathsf{b},\ \mathsf{f}(\mathsf{c}(x,y),z) \to \mathsf{c}(\mathsf{f}(x,y),\mathsf{f}(z,x)) \}$. A narrowing tree for the pair $\mathsf{f}(x,y) \twoheadrightarrow v$ is illustrated in Figure 1. Labeled solid arrows "$\xrightarrow{\theta}$" represent innermost-narrowing steps with relevant substitutions θ,[3] double-line arcs "$=$" decompose nests of defined symbols (*flattening*), double arrows "$\Longrightarrow$" divide equations to single ones (*splitting*), labeled dotted arrows "$\overset{\delta}{\dashrightarrow}$" visualize the existence of a variant node connected via a *renaming*[4] δ (*recursion*). The narrowing tree in Figure 1 can be written by the following *grammar representation* [29] that can be considered a *regular tree grammar* [3]:

$$
\begin{aligned}
\Gamma_{\mathsf{f}(x,y)\twoheadrightarrow v} \to\ & \{v \mapsto \mathsf{a}\} \bullet \{x \mapsto \mathsf{a}\} \mid \{v \mapsto \mathsf{b}\} \bullet \{x \mapsto \mathsf{b}\} \\
& \mid
\begin{pmatrix}
\Gamma_{\mathsf{f}(x,y)\twoheadrightarrow v} \bullet \{x' \mapsto x,,\ y' \mapsto y,\ v' \mapsto v\} \\
\& \\
\Gamma_{\mathsf{f}(x,y)\twoheadrightarrow v} \bullet \{x' \mapsto y,\ z' \mapsto x,\ v'' \mapsto v\} \\
\& \\
\{v \mapsto \mathsf{c}(v',v'')\}
\end{pmatrix} \bullet \{x \mapsto \mathsf{c}(x',y'),\ y \mapsto z'\}
\end{aligned}
\tag{1}
$$

The binary symbols $\bullet$ and $\&$ are interpreted by standard composition and *parallel composition* [13, 33], respectively. Parallel composition $\Uparrow$ of two idempotent substitutions returns a most general unifier of the substitutions if the substitutions are unifiable. For example, $\{y' \mapsto \mathsf{a},\ y \mapsto \mathsf{a}\} \Uparrow \{y' \mapsto y\}$ returns $\{y' \mapsto \mathsf{a},\ y \mapsto \mathsf{a}\}$ and $\{y' \mapsto \mathsf{a},\ y \mapsto \mathsf{b}\} \Uparrow \{y' \mapsto y\}$ fails. Due to parallel composition (i.e., occurrence of $\&$), it is not so easy to not only analyze but also simplify grammar representations of narrowing trees. In the remaining of this paper, we do not deal with narrowing trees but their grammar representations.

Throughout this paper, we aim at proving infeasibility of the condition $\mathsf{o}(x) \twoheadrightarrow \mathsf{true}\ \&$ $\mathsf{e}(x) \twoheadrightarrow \mathsf{true}$ for $\mathcal{R}_1$[5] w.r.t. constructor-based rewriting. To this end, we first show that every

---

[3] One may think that y of $\mathsf{f}(x,y) \twoheadrightarrow v$ in Figure 1 does not have to be instantiated by z' because y is received by a variable that can be seen as patternless. However, the tree is used two or more times via dotted arrows, and the reuse always starts with $\mathsf{f}(x,y) \twoheadrightarrow v$ that is connected by means of a renaming attached with dotted arrows. To avoid any conflict of using y, we always introduce only fresh variables at narrowing steps, i.e., $\mathsf{f}(x,y) \overset{i}{\rightsquigarrow}_{\mathcal{R}_2} \mathsf{c}(\mathsf{f}(x',y),\mathsf{f}(z',x'))$ is not allowed (see Definition 3 in Section 3).

[4] To be precise, δ (e.g., $\{x' \mapsto y,\ z' \mapsto x,\ v'' \mapsto v\}$ in Figure 1) is not a renaming, while we can write an exact renaming. However, we write such a substitution, so-called a *prenaming* [17], obtained by restricting a renaming to variables that we are interested in because the renaming is used to rename a particular term.

[5] We use $\mathcal{R}_1$, which is an SDCTRS but also a normal 1-CTRS, as a leading example of this paper because $\mathcal{R}_1$ is reasonable to illustrate how we can use the grammar representation of a narrowing tree to prove confluence of a CTRS.

constructor SDCTRS can be converted to an equivalent unconditional constructor system which may have extra variables (Section 3). Secondly, we revisit *compositionality* of innermost narrowing, relaxing some assumptions in [29] (Section 4). Thirdly, we introduce grammar representations of sets of idempotent substitutions as regular tree grammars (Section 5) and a construction of narrowing trees for given unconditional constructor systems (Section 6). Fourthly, we show some methods to simplify grammar representations of narrowing trees (Section 7). Finally, we show that grammar representations of narrowing trees are useful to prove infeasibility of conditional critical pairs of $\mathcal{R}_1$ and *quasi-reducibility* [16] of $\mathcal{R}_1$ with usual sorts for natural numbers and boolean values (Section 8). Quasi-reducibility is that every ground basic term is defined (i.e., reducible). For (operationally) terminating (C)TRSs, quasi-reducibility is equivalent to *sufficient completeness* (cf. [15, 2]). The results in this paper would straightforwardly be extended to *many sorted* systems. Differences to related work are described in Section 9, and proofs of theorems are shown in Appendix B.

The contribution of this paper is to show a method that can prove (1) confluence of $\mathcal{R}_1$, for which all existing confluence provers other than AGES fail to prove confluence, and (2) quasi-reducibility of $\mathcal{R}_1$.

2   Preliminaries

In this section, we recall basic notions and notations of term rewriting [1, 32] and regular tree grammars [3].

Throughout the paper, we use $\mathcal{V}$ as a countably infinite set of *variables*. Let $\mathcal{F}$ be a *signature*, a finite set of *function symbols* f each of which has its own fixed arity. We often write $f/n \in \mathcal{F}$ instead of "an n-ary symbol $f \in \mathcal{F}$", and so on. The set of *terms* over $\mathcal{F}$ and V ($\subseteq \mathcal{V}$) is denoted by $\mathcal{T}(\mathcal{F}, V)$, and $\mathcal{T}(\mathcal{F}, \emptyset)$, the set of *ground* terms, is abbreviated to $\mathcal{T}(\mathcal{F})$. The set of variables appearing in any of terms $t_1, \ldots, t_n$ is denoted by $\mathcal{V}ar(t_1, \ldots, t_n)$. For a term t and a position p of t, the *subterm* of t at p is denoted by $t|_p$. Given terms s, t and a position p of s, we denote by $s[t]_p$ the term obtained from s by replacing the subterm $s|_p$ at p by t.

A *substitution* σ is a mapping from variables to terms such that the number of variables x with $\sigma(x) \neq x$ is finite, and is naturally extended over terms. The *domain* and *range* of σ are denoted by $\mathcal{D}om(\sigma)$ and $\mathcal{R}an(\sigma)$, respectively. The set of variables in $\mathcal{R}an(\sigma)$ is denoted by $\mathcal{V}\mathcal{R}an(\sigma)$. We may denote σ by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ if $\mathcal{D}om(\sigma) = \{x_1, \ldots, x_n\}$ and $\sigma(x_i) = t_i$ for all $1 \leq i \leq n$. The *identity* substitution is denoted by *id*. The set of substitutions that range over a signature $\mathcal{F}$ and a set V of variables is denoted by $Subst(\mathcal{F}, V)$: $Subst(\mathcal{F}, V) = \{\sigma \mid \sigma$ is a substitution, $\mathcal{R}an(\sigma) \subseteq \mathcal{T}(\mathcal{F}, V)\}$. The application of a substitution σ to a term t is abbreviated to σt, and σt is called an *instance* of t. Given a set V of variables, $\sigma|_V$ denotes the *restricted* substitution of σ w.r.t. V: $\sigma|_V = \{x \mapsto \sigma x \mid x \in \mathcal{D}om(\sigma) \cap V\}$. A substitution σ is called a *renaming* if σ is a bijection on $\mathcal{V}$. The *composition* $\theta \cdot \sigma$ (simply $\theta\sigma$) of substitutions σ and θ is defined as $(\theta \cdot \sigma)(x) = \theta(\sigma(x))$. A substitution σ is called *idempotent* if $\sigma\sigma = \sigma$ (i.e., $\mathcal{D}om(\sigma) \cap \mathcal{V}\mathcal{R}an(\sigma) = \emptyset$). A substitution σ is called *more general than* a substitution θ, written by $\sigma \leq \theta$, if there exists a substitution δ such that $\delta\sigma = \theta$. A finite set E of term equations $s \approx t$ is called *unifiable* if there exists a *unifier* of E such that $\sigma s = \sigma t$ for all term equations $s \approx t$ in E. A *most general unifier* (mgu, for short) of E is denoted by $mgu(E)$ if E is unifiable. Terms s and t are called *unifiable* if $\{s \approx t\}$ is unifiable. The application of a substitution θ to E is defined as $\theta(E) = \{\theta s \approx \theta t \mid s \approx t \in E\}$.

An *oriented conditional rewrite rule* over a signature $\mathcal{F}$ is a triple (ℓ, r, c), denoted by $\ell \to r \Leftarrow c$, such that the *left-hand side* ℓ is a non-variable term in $\mathcal{T}(\mathcal{F}, \mathcal{V})$, the *right-hand*

side r is a term in $\mathcal{T}(\mathcal{F}, \mathcal{V})$, and the *conditional part c* is a sequence $s_1 \twoheadrightarrow t_1, \ldots, s_k \twoheadrightarrow t_k$ of term pairs ($k \geq 0$) where $s_1, t_1, \ldots, s_k, t_k \in \mathcal{T}(\mathcal{F}, \mathcal{V})$. In particular, a conditional rewrite rule is called *unconditional* if the conditional part is the empty sequence (i.e., $k = 0$), and we may abbreviate it to $\ell \to r$. Variables in $\mathcal{V}ar(r, c) \setminus \mathcal{V}ar(\ell)$ are called *extra variables* of the rule. An *oriented conditional term rewriting system* (a CTRS, for short) over $\mathcal{F}$ is a set of oriented conditional rewrite rules over $\mathcal{F}$. A CTRS is called an (unconditional) *term rewriting system* (TRS) if every rule $\ell \to r \Leftarrow c$ in the CTRS is unconditional and satisfies $\mathcal{V}ar(\ell) \supseteq \mathcal{V}ar(r)$. The *reduction relation* $\to_{\mathcal{R}}$ of a CTRS $\mathcal{R}$ is defined as $\to_{\mathcal{R}} = \bigcup_{n \geq 0} \to_{(n), \mathcal{R}}$, where $\to_{(0), \mathcal{R}} = \emptyset$, and $\to_{(i+1), \mathcal{R}} = \{(s[\sigma \ell]_p, s[\sigma r]_p) \mid s \in \mathcal{T}(\mathcal{F}, \mathcal{V}),\ \ell \to r \Leftarrow s_1 \twoheadrightarrow t_1, \ldots, s_k \twoheadrightarrow t_k \in \mathcal{R},\ \sigma s_1 \to_{(i), \mathcal{R}}^* \sigma t_1,\ \ldots,\ \sigma s_k \to_{(i), \mathcal{R}}^* \sigma t_k\}$ for $i \geq 0$. To specify the position where the rule is applied, we may write $\to_{p, \mathcal{R}}$ instead of $\to_{\mathcal{R}}$. The *underlying unconditional system* $\{\ell \to r \mid \ell \to r \Leftarrow c \in \mathcal{R}\}$ of $\mathcal{R}$ is denoted by $\mathcal{R}_u$. A term t is called a *normal form* (of $\mathcal{R}$) if t is irreducible w.r.t. $\mathcal{R}$. A substitution σ is called *normalized* (w.r.t. $\mathcal{R}$) if σx is a normal form of $\mathcal{R}$ for each variable $x \in \mathcal{D}om(\sigma)$. A CTRS $\mathcal{R}$ is called *Type 1* (1-CTRS, for short) if every rule $\ell \to r \Leftarrow c \in \mathcal{R}$ satisfies that $\mathcal{V}ar(r, c) \subseteq \mathcal{V}ar(\ell)$; *Type 3* (3-CTRS, for short) if every rule $\ell \to r \Leftarrow c \in \mathcal{R}$ satisfies that $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(\ell, c)$; *normal* if for every rule $\ell \to r \Leftarrow s_1 \twoheadrightarrow t_1, \ldots, s_k \twoheadrightarrow t_k \in \mathcal{R}$, all $t_1, \ldots, t_k$ are ground normal forms of $\mathcal{R}_u$; *deterministic* (a DCTRS, for short) if, for every rule $\ell \to r \Leftarrow s_1 \twoheadrightarrow t_1, \ldots, s_k \twoheadrightarrow t_k \in \mathcal{R}$, $\mathcal{V}ar(s_i) \subseteq \mathcal{V}ar(\ell, t_1, \ldots, t_{i-1})$ for all $1 \leq i \leq k$.

The sets of *defined symbols* and *constructors* of a CTRS $\mathcal{R}$ over a signature $\mathcal{F}$ are denoted by $\mathcal{D}_{\mathcal{R}}$ and $\mathcal{C}_{\mathcal{R}}$, respectively: $\mathcal{D}_{\mathcal{R}} = \{f \mid f(u_1, \ldots, u_n) \to r \Leftarrow c \in \mathcal{R}\}$ and $\mathcal{C}_{\mathcal{R}} = \mathcal{F} \setminus \mathcal{D}_{\mathcal{R}}$. Terms in $\mathcal{T}(\mathcal{C}_{\mathcal{R}}, \mathcal{V})$ are called *constructor terms of $\mathcal{R}$*. A substitution in $Subst(\mathcal{C}_{\mathcal{R}}, \mathcal{V})$ is called a *constructor substitution of $\mathcal{R}$*. A term of the form $f(t_1, \ldots, t_n)$ with $f/n \in \mathcal{D}_{\mathcal{R}}$ and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C}_{\mathcal{R}}, \mathcal{V})$ is called *basic*. A CTRS $\mathcal{R}$ is called a *constructor system* if for every rule $\ell \to r \Leftarrow c$ in $\mathcal{R}$, ℓ is basic. A CTRS $\mathcal{R}$ is called a *pure-constructor system* (a pc-CTRS, for short) if for every rule $\ell \to r \Leftarrow s_1 \twoheadrightarrow t_1, \ldots, s_k \twoheadrightarrow t_k \in \mathcal{R}$, all of $\ell, s_1, \ldots, s_k$ are basic and all of $r, t_1, \ldots, t_k$ are constructor terms [24]. A 3-DCTRS $\mathcal{R}$ is called *syntactically deterministic* (an SDCTRS, for short) if for every rule $\ell \to r \Leftarrow s_1 \twoheadrightarrow t_1, \ldots, s_k \twoheadrightarrow t_k \in \mathcal{R}$, every t_i is a constructor term or a ground normal form of $\mathcal{R}_u$.

A CTRS $\mathcal{R}$ is called *operationally terminating* if there are no infinite well-formed trees in a certain logical inference system [21] – operational termination means that the evaluation of conditions must either successfully terminate or fail in finite time. Two terms s and t are said to be *joinable*, written as $s \downarrow_{\mathcal{R}} t$, if there exists a term u such that $s \to_{\mathcal{R}}^* u \leftarrow_{\mathcal{R}}^* t$. A CTRS $\mathcal{R}$ is called *confluent* if $t_1 \downarrow_{\mathcal{R}} t_2$ for any terms t_1, t_2 with $t_1 \leftarrow_{\mathcal{R}}^* \cdot \to_{\mathcal{R}}^* t_2$.

A *regular tree grammar* is a quadruple $\mathcal{G} = (S, \mathcal{N}, \mathcal{F}, \mathcal{P})$ such that $\mathcal{F}$ is a signature, $\mathcal{N}$ is a finite set of *non-terminals* (constants not in $\mathcal{F}$), $S \in \mathcal{N}$, and $\mathcal{P}$ is a finite set of *production rules* of the form $A \to \beta$ with $A \in \mathcal{N}$ and $\beta \in \mathcal{T}(\mathcal{F} \cup \mathcal{N})$. Note that $A \to \beta_1 \mid \ldots \mid \beta_n$ stands for $A \to \beta_1, \ldots, A \to \beta_n$. Given a non-terminal $S' \in \mathcal{N}$, the set $\{t \in \mathcal{T}(\mathcal{F}) \mid S' \to_{\mathcal{P}}^* t\}$ is the *language generated by $\mathcal{G}$ from S'*, denoted by $L(\mathcal{G}, S')$. The *initial* non-terminal S does not play an important role in this paper. A *regular tree language* is a language generated by a regular tree grammar from one of its non-terminals. The class of regular tree languages is equivalent to the class of *recognizable tree languages* which are recognized by *tree automata*. This means that the *intersection (non-)emptiness problem* for regular tree languages is decidable.

▶ **Example 1.** The regular tree grammar $\mathcal{G}_1 = (X, \{X, X'\}, \{0/0, s/1\}, \{X \to 0 \mid s(X'), X' \to s(X)\})$ generates the sets of even and odd numbers over 0 and s from X and X', respectively: $L(\mathcal{G}_1, X) = \{s^{2n}(0) \mid n \geq 0\}$ ($= L(\mathcal{G}_1)$) and $L(\mathcal{G}_1, X') = \{s^{2n+1}(0) \mid n \geq 0\}$.

3   From Constructor SDCTRSs to Unconditional Constructor Systems

In this section, we show that every constructor SDCTRS can be converted to an equivalent unconditional constructor system w.r.t. *constructor-based rewriting* and *innermost narrowing* for *goal clauses*. Since SDCTRSs possibly have extra variables, we relax the requirement "$\mathcal{V}ar(\ell) \supseteq \mathcal{V}ar(r)$" for TRSs, i.e., we allow unconditional rules to have extra variables.

We denote a pair of terms s, t by $s \twoheadrightarrow t$ (not an equation $s \approx t$) because we analyze conditions of rewrite rules and distinguish the left- and right-hand sides of the pair $s \twoheadrightarrow t$. We deal with pairs of terms as terms by considering $\twoheadrightarrow$ a binary function symbol. For this reason, we apply many notions for terms to pairs of terms without notice. For readability, when we deal with $s \twoheadrightarrow t$ as a term, we often put it in parentheses: $(s \twoheadrightarrow t)$. As in [23], we assume that any CTRS in this paper implicitly includes the rule $(x \twoheadrightarrow x) \to \top$ where $\top$ is a special constant. The rule $(x \twoheadrightarrow x) \to \top$ is used to test equivalence between two terms t_1, t_2 via $t_1 \twoheadrightarrow t_2$. A pair $s \twoheadrightarrow t$ of terms s, t is called a *goal* of a constructor SDCTRS $\mathcal{R}$ if the left-hand side s is either a constructor term or a basic term and the right-hand side t is a constructor term.

To deal with a conjunction of pairs $e_1, \ldots, e_k$ of terms (e_i is either $s_i \twoheadrightarrow t_i$ or $\top$) as a term, we write $e_1 \,\&\, \cdots \,\&\, e_k$ by using an associative binary symbol $\&$. We call such a term an *equational term*. Unlike [29], to avoid $\&$ to be a defined symbol, we do not use any rule for $\&$, e.g., $(\top \,\&\, x) \to x$. Instead of derivations ending with $\top$, we consider derivations that end with terms in $\mathcal{T}(\{\top, \&\})$. We assume that none of $\&$, $\twoheadrightarrow$, or $\top$ is included in the range of any substitution below. In the following, we denote conditional parts of rules by equational terms, e.g., $\ell \to r \Leftarrow s_1 \twoheadrightarrow t_1 \,\&\, \cdots \,\&\, s_k \twoheadrightarrow t_k$. Note that the empty sequence of a conditional part is denoted by $\top$. An equational term is called a *goal clause* of a constructor SDCTRS $\mathcal{R}$ if it is a conjunction of goals for $\mathcal{R}$. Note that for a goal clause T, any instance θT with θ a constructor substitution is a goal clause.

▶ **Example 2.** The equational term $\mathsf{e}(x) \twoheadrightarrow \mathsf{true} \,\&\, \mathsf{o}(x) \twoheadrightarrow \mathsf{true}$ is a goal clause of $\mathcal{R}_1$.

3.1   Constructor-based Rewriting and Innermost Narrowing

Following [28], we define *constructor-based conditional rewriting* on goal clauses as follows: for a goal clause $S = U \,\&\, s \twoheadrightarrow t \,\&\, S'$ with $U \in \mathcal{T}(\{\top, \&\})$, we write $S \xrightarrow{c}_{\mathcal{R}} T$ if there exist a non-variable position p of $(s \twoheadrightarrow t)$, a rule $\ell \to r \Leftarrow C$ in $\mathcal{R}$, and a constructor substitution σ such that $(s \twoheadrightarrow t)|_p$ is basic, $(s \twoheadrightarrow t)|_p = \sigma\ell$, and $T = U \,\&\, \sigma C \,\&\, (s \twoheadrightarrow t)[\sigma r]_p \,\&\, S'$. The constructor-based rewriting under the leftmost strategy is denoted by $\xrightarrow{lc}_{\mathcal{R}}$. It is clear that for a goal clause S and a normal form T of $\mathcal{R}$, $S \xrightarrow{c}{}^*_{\mathcal{R}} T$ if and only if $S \xrightarrow{lc}{}^*_{\mathcal{R}} T$.

The *narrowing* relation [34, 14] mainly extends rewriting by replacing matching with unification. This paper follows the formalization in [28], while we use the rule $(x \twoheadrightarrow x) \to \top$ instead of the corresponding inference rule.

▶ **Definition 3** (innermost narrowing). Let $\mathcal{R}$ be a CTRS. A goal clause $S = U \,\&\, s \twoheadrightarrow t \,\&\, S'$ with $U \in \mathcal{T}(\{\top, \&\})$ is said to *conditionally narrow* into an equational term T at an innermost position, written as $S \overset{i}{\rightsquigarrow}_{\mathcal{R}} T$, if there exist a non-variable position p of $(s \twoheadrightarrow t)$, a variant $\ell \to r \Leftarrow C$ of a rule in $\mathcal{R}$, and a constructor substitution σ such that $\mathcal{V}ar(\ell, r, C) \cap \mathcal{V}ar(S) = \emptyset$, $(s \twoheadrightarrow t)|_p$ is basic, $(s \twoheadrightarrow t)|_p$ and ℓ are unifiable, $\sigma = mgu(\{(s \twoheadrightarrow t)|_p \approx \ell\})$, and $T = U \,\&\, \sigma C \,\&\, \sigma((s \twoheadrightarrow t)[r]_p) \,\&\, \sigma S'$. Note that all extra variables of $\ell \to r \Leftarrow C$ remain in T as *fresh* variables which do not appear in S. We assume that $\mathcal{V}ar(S) \cap \mathcal{V}\mathcal{R}an(\sigma|_{\mathcal{V}ar((s \twoheadrightarrow t)|_p)}) = \emptyset$ (i.e., $\sigma|_{\mathcal{V}ar((s \twoheadrightarrow t)|_p)}$ is idempotent) and $\mathcal{V}ar((s \twoheadrightarrow t)|_p) \subseteq \mathcal{D}om(\sigma)$. We write $S \overset{li}{\rightsquigarrow}_{\mathcal{R}} T$ if p is the leftmost among innermost narrowable positions in $(s \twoheadrightarrow t)$. We write $S \overset{i}{\rightsquigarrow}_{\sigma|_{\mathcal{V}ar(S)}, \mathcal{R}} T$ to make the substitution explicit.

An example of innermost narrowing and constructor-based rewriting can be seen in Appendix A. Let $\overset{x}{\rightsquigarrow}_{\mathcal{R}}$ be either $\overset{i}{\rightsquigarrow}_{\mathcal{R}}$ or $\overset{li}{\rightsquigarrow}_{\mathcal{R}}$. An *innermost narrowing derivation* $T_0 \overset{x}{\rightsquigarrow}{}^*_{\sigma,\mathcal{R}} T_n$ (and $T_0 \overset{x}{\rightsquigarrow}{}^n_{\sigma,\mathcal{R}} T_n$) denotes a sequence of narrowing steps $T_0 \overset{x}{\rightsquigarrow}_{\sigma_1,\mathcal{R}} \cdots \overset{x}{\rightsquigarrow}_{\sigma_n,\mathcal{R}} T_n$ with $\sigma = (\sigma_n \cdots \sigma_1)|_{\mathcal{V}ar(T_0)}$ an idempotent substitution. When we consider two (or more) narrowing derivations $S_1 \overset{x}{\rightsquigarrow}{}^*_{\sigma_1,\mathcal{R}} T_1$ and $S_2 \overset{x}{\rightsquigarrow}{}^*_{\sigma_2,\mathcal{R}} T_2$, we assume that $\mathcal{VR}an(\sigma_1) \cap \mathcal{VR}an(\sigma_2) = \emptyset$.

As in [29], for the sake of simplicity, we first consider the leftmost innermost narrowing. After showing basic properties of compositionality, we drop this restriction (see Theorem 14).

Constructor-based rewriting and innermost narrowing of constructor SDCTRSs have the following relationships (cf. [28]).

▶ **Theorem 4.** *Let $\mathcal{R}$ be a constructor SDCTRS, T a goal clause, and $U \in \mathcal{T}(\{\top, \&\})$.*

1. *If $T \overset{li}{\rightsquigarrow}{}^*_{\sigma,\mathcal{R}} U$, then $\sigma T \overset{lc}{\rightarrow}{}^*_{\mathcal{R}} U$ (i.e., $\sigma s \overset{lc}{\rightarrow}{}^*_{\mathcal{R}} \sigma t$ for all goals $s \twoheadrightarrow t$ in T).*

2. *For a constructor substitution θ, if $\theta T \overset{lc}{\rightarrow}{}^*_{\mathcal{R}} U$, then there exists an idempotent constructor substitution σ such that $T \overset{li}{\rightsquigarrow}{}^*_{\sigma,\mathcal{R}} U$ and $\sigma \leq \theta$.*

3.2 Converting to Unconditional Constructor Systems

We say that a constructor SDCTRS $\mathcal{R}$ over a signature $\mathcal{F}$ is *equivalent* to a constructor SDCTRS $\mathcal{R}'$ over $\mathcal{F}$ w.r.t. $\overset{c}{\rightarrow}$ and $\overset{i}{\rightsquigarrow}$ if $\mathcal{D}_{\mathcal{R}} = \mathcal{D}_{\mathcal{R}'}$ and both of the following hold:

- for any goal clause T, $T \overset{lc}{\rightarrow}{}^*_{\mathcal{R}} U$ for some term $U \in \mathcal{T}(\{\top, \&\})$ if and only if $T \overset{lc}{\rightarrow}{}^*_{\mathcal{R}'} U'$ for some term $U' \in \mathcal{T}(\{\top, \&\})$, and
- for any goal clause T, $T \overset{li}{\rightsquigarrow}{}^*_{\theta,\mathcal{R}} U$ for some term $U \in \mathcal{T}(\{\top, \&\})$ if and only if $T \overset{li}{\rightsquigarrow}{}^*_{\theta,\mathcal{R}'} U'$ for some term $U' \in \mathcal{T}(\{\top, \&\})$.

Note that $\mathcal{C}_{\mathcal{R}} = \mathcal{C}_{\mathcal{R}'}$. In this section, we first convert a constructor SDCTRS to a pc-CTRS that is equivalent to the SDCTRS w.r.t. $\overset{c}{\rightarrow}$ and $\overset{i}{\rightsquigarrow}$, and then convert the pc-CTRS to a constructor TRS that is equivalent to the pc-CTRS w.r.t. $\overset{c}{\rightarrow}$ and $\overset{i}{\rightsquigarrow}$.

To convert a constructor SDCTRS $\mathcal{R}$, we adopt a stepwise transformation for each rule $\ell \rightarrow r \Leftarrow C \in \mathcal{R}$ as follows (cf. [26, Definition 23]).

▶ **Definition 5.** We transform each rule $\ell \rightarrow r \Leftarrow C$ of a constructor SDCTRS $\mathcal{R}$ as follows:

1. We replace r and C by a fresh variable y and $C, r \twoheadrightarrow y$, respectively, if $r \notin \mathcal{T}(\mathcal{C}_{\mathcal{R}}, \mathcal{V})$.

2. For each condition $s \twoheadrightarrow t$ in the resulting conditional part, if t contains a defined symbol, then we replace $s \twoheadrightarrow t$ by $(s \twoheadrightarrow x) \& (t \twoheadrightarrow x)$, where x is a fresh variable.[6]

3. We remove all nests of defined symbols in the resulting conditional part by replacing a condition $s[f(u_1, \ldots, u_n)]_p \twoheadrightarrow t$ with $(f(u_1, \ldots, u_n) \twoheadrightarrow x) \& (s[x]_p \twoheadrightarrow t)$, where f is a defined symbol, $p > \varepsilon$, and x is a fresh variable that does not appear in the intermediate rule. This operation is so-called a *flattening* [29] shown in Section 4.

4. If the resulting rule has a condition $s \twoheadrightarrow t$ with s, t constructor terms, then (1) we drop the rule from $\mathcal{R}$ whenever s and t are not unifiable, and (2) otherwise, we drop the condition $s \twoheadrightarrow t$ by applying an *mgu* of s, t to the rule [27, p. 292] (see also [26, Theorem 26]).

We denote the resulting CTRS by $Pc(\mathcal{R})$.

---

[6] If C contains a condition $s \twoheadrightarrow t$ such that t contains a defined symbol, then rule $\ell \rightarrow r \Leftarrow C$ is never used in constructor-based rewriting of goal clauses to terms in $\mathcal{T}(\{\top, \&\})$ because t is not a constructor term and any instance of $s \twoheadrightarrow t$ is never reduced to any term in $\mathcal{T}(\{\top, \&\})$. However, we do not drop the rule from $\mathcal{R}$ because defined symbols are preserved during the conversion and the rule can be used for the standard rewriting $\rightarrow_{\mathcal{R}}$.

▶ **Theorem 6.** *Let $\mathcal{R}$ be a constructor SDCTRS over a signature $\mathcal{F}$. Then, $Pc(\mathcal{R})$ is a pc-CTRS over $\mathcal{F}$ and is equivalent to $\mathcal{R}$ w.r.t. $\xrightarrow{c}$ and $\overset{i}{\rightsquigarrow}$.*

Let $\mathcal{R}$ be a pc-CTRS over $\mathcal{F}$. We denote the TRS $\{(\ell \twoheadrightarrow y) \to C \ \& \ (r \twoheadrightarrow y) \mid \ell \to r \Leftarrow C \in \mathcal{R}, \ y \in \mathcal{V} \setminus \mathcal{V}ar(\ell, r, C)\}$ by $Trs(\mathcal{R})$. Since the conditional part is a goal clause, the generated right-hand side $C \ \& \ (r \twoheadrightarrow y)$ is a goal clause. Thus, for a goal clause T, if $T \xrightarrow{c}_{\mathcal{R}} T'$ or $T \overset{i}{\rightsquigarrow}_{\mathcal{R}} T'$, then T' is a goal clause. It is clear that $Trs(\mathcal{R})$ is a constructor TRS, $\mathcal{D}_{Trs(\mathcal{R})} = \{\twoheadrightarrow\}$, and $\mathcal{C}_{Trs(\mathcal{R})} = \mathcal{F} \cup \{\top, \&\}$.

▶ **Theorem 7.** *Let $\mathcal{R}$ be a pc-CTRS over a signature $\mathcal{F}$. Then, $Trs(\mathcal{R})$ is a constructor TRS over $\mathcal{F} \cup \{\twoheadrightarrow, \top, \&\}$ and is equivalent to $\mathcal{R}$ w.r.t. $\xrightarrow{c}$ and $\overset{i}{\rightsquigarrow}$.*

▶ **Example 8.** For $\mathcal{R}_1$ in Section 1, we obtain the following TRS by applying $Trs(\cdot)$ to $\mathcal{R}_1$:

$$Trs(\mathcal{R}_1) = \left\{ \begin{array}{ll} (\mathsf{e}(0) \twoheadrightarrow y) \to (\mathsf{true} \twoheadrightarrow y), & (\mathsf{e}(\mathsf{s}(x)) \twoheadrightarrow y) \to (\mathsf{o}(x) \twoheadrightarrow \mathsf{true}) \ \& \ (\mathsf{true} \twoheadrightarrow y), \\ & (\mathsf{e}(\mathsf{s}(x)) \twoheadrightarrow y) \to (\mathsf{e}(x) \twoheadrightarrow \mathsf{true}) \ \& \ (\mathsf{false} \twoheadrightarrow y), \\ (\mathsf{o}(0) \twoheadrightarrow y) \to (\mathsf{false} \twoheadrightarrow y), & (\mathsf{o}(\mathsf{s}(x)) \twoheadrightarrow y) \to (\mathsf{e}(x) \twoheadrightarrow \mathsf{true}) \ \& \ (\mathsf{true} \twoheadrightarrow y), \\ & (\mathsf{o}(\mathsf{s}(x)) \twoheadrightarrow y) \to (\mathsf{o}(x) \twoheadrightarrow \mathsf{true}) \ \& \ (\mathsf{false} \twoheadrightarrow y) \end{array} \right\}$$

For example, the following narrowing derivation holds for both $\mathcal{R}_1$ and $Trs(\mathcal{R}_1)$: $(\mathsf{e}(x) \twoheadrightarrow \mathsf{true}) \overset{li}{\rightsquigarrow}_{\{x \mapsto \mathsf{s}(x_1)\}} (\mathsf{o}(x_1) \twoheadrightarrow \mathsf{true}) \ \& \ (\mathsf{true} \twoheadrightarrow \mathsf{true}) \overset{li}{\rightsquigarrow}_{\{x_1 \mapsto 0\}} (\mathsf{false} \twoheadrightarrow \mathsf{true}) \ \& \ (\mathsf{true} \twoheadrightarrow \mathsf{true})$.

As a consequence of Theorems 6 and 7, we obtain the following corollary.

▶ **Corollary 9.** *Let $\mathcal{R}$ be a constructor SDCTRS over a signature $\mathcal{F}$. Then, $Trs(Pc(\mathcal{R}))$ is a constructor TRS over $\mathcal{F} \cup \{\twoheadrightarrow, \top, \&\}$ and is equivalent to $\mathcal{R}$ w.r.t. $\xrightarrow{c}$ and $\overset{i}{\rightsquigarrow}$.*

4   Compositionality of Innermost Narrowing

Compositionality of innermost narrowing under *parallel composition* of idempotent substitutions is a key to ensure equivalence between substitutions obtained at innermost-narrowing steps and those defined by grammar representations of narrowing trees. In this section, we recall parallel composition, and revisit compositionality of innermost narrowing for TRSs. Since the counterpart of $\overset{i}{\rightsquigarrow}_{\mathcal{R}}$ is constructor-based rewriting $\xrightarrow{c}_{\mathcal{R}}$, sufficient completeness is required in [29] to have $\xrightarrow{c}_{\mathcal{R}} = \xrightarrow{i}_{\mathcal{R}}$ on ground terms. However, sufficient completeness is not necessary for compositionality and we do not force this property to TRSs.

We first recall *parallel composition* $\Uparrow$ of idempotent substitutions [13, 33], which is one of the most important key operations to enable us to construct *finite* narrowing trees. Given a substitution $\theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, we denote the set of term equations $\{x_1 \approx t_1, \ldots, x_n \approx t_n\}$ by $\widehat{\theta}$.

▶ **Definition 10** (parallel composition $\Uparrow$ [33])**.** Let θ_1 and θ_2 be idempotent substitutions. Then, we define $\Uparrow$ as follows: $\theta_1 \Uparrow \theta_2 = mgu(\widehat{\theta_1} \cup \widehat{\theta_2})$ if $\widehat{\theta_1} \ \& \ \widehat{\theta_2}$ is unifiable, and otherwise, $\theta_1 \Uparrow \theta_2 = fail$. Note that we define $\theta_1 \Uparrow \theta_2 = fail$ if θ_1 or θ_2 is not idempotent. Parallel composition is extended to sets Θ_1, Θ_2 of idempotent substitutions in the natural way: $\Theta_1 \Uparrow \Theta_2 = \{\theta_1 \Uparrow \theta_2 \mid \theta_1 \in \Theta_1, \ \theta_2 \in \Theta_2, \ \theta_1 \Uparrow \theta_2 \neq fail\}$.

We often have two or more substitutions that can be results of $\theta_1 \Uparrow \theta_2 \ (\neq fail)$, while they are unique up to variable renaming. To simplify the semantics of grammar representations for substitutions, we adopt an idempotent substitution σ with $\mathcal{D}om(\theta_1) \cup \mathcal{D}om(\theta_2) \subseteq \mathcal{D}om(\sigma)$ as a result of $\theta_1 \Uparrow \theta_2 \ (\neq fail)$. Idempotent substitutions we can adopt as results of $\theta_1 \Uparrow \theta_2$ under the convention are unique up to variable renaming, but not exactly unique in general.

▶ **Example 11.** The parallel composition $\{x \mapsto \mathsf{s}(z),\, y \mapsto z\} \Uparrow \{x \mapsto w\}$ may return $\{x \mapsto \mathsf{s}(z),\, y \mapsto z,\, w \mapsto \mathsf{s}(z)\}$, but we do not allow $\{x \mapsto \mathsf{s}(y),\, z \mapsto y,\, w \mapsto \mathsf{s}(y)\}$ as a result. On the other hand, $\{x \mapsto \mathsf{s}(z),\, y \mapsto z\} \Uparrow \{x \mapsto y\}$ fails.

Let $\overset{x}{\rightsquigarrow}_{\mathcal{R}}$ be either $\overset{i}{\rightsquigarrow}_{\mathcal{R}}$ or $\overset{li}{\rightsquigarrow}_{\mathcal{R}}$. For a constructor SDCTRS $\mathcal{R}$ and a goal clause T, we define the *success set* of T (w.r.t. $\overset{x}{\rightsquigarrow}_{\mathcal{R}}$), which is the set of *successful* substitutions derived by $\overset{x}{\rightsquigarrow}_{\mathcal{R}}$, as follows: $Suc(\overset{x}{\rightsquigarrow}_{\mathcal{R}}, T) = \{\theta \mid \exists U \in \mathcal{T}(\{\top, \&\}).\ T \overset{x}{\rightsquigarrow}{}^{*}_{\theta, \mathcal{R}} U\}$. Note that $\mathcal{D}om(\theta) \subseteq \mathcal{V}ar(T)$ for any substitution $\theta \in Suc(\overset{x}{\rightsquigarrow}_{\mathcal{R}}, T)$. We extend the restriction of substitutions to sets of substitutions: $\Theta|_V = \{\theta|_V \mid \theta \in \Theta\}$.

▶ **Theorem 12** (compositionality [29]). *For a constructor TRS $\mathcal{R}$ and goal clauses T_1, T_2,* $Suc(\overset{li}{\rightsquigarrow}_{\mathcal{R}}, T_1 \& T_2) = \left(Suc(\overset{li}{\rightsquigarrow}_{\mathcal{R}}, T_1) \Uparrow Suc(\overset{li}{\rightsquigarrow}_{\mathcal{R}}, T_2) \right)|_{\mathcal{V}ar(T_1, T_2)}$ *up to variable renaming.*

Note that $\&$ is just a binary symbol to construct conjunctions of goals, and $\Uparrow$ is a binary operator for parallel composition. In Theorem 12, we restrict $Suc(\overset{li}{\rightsquigarrow}_{\mathcal{R}}, T_1) \Uparrow Suc(\overset{li}{\rightsquigarrow}_{\mathcal{R}}, T_2)$ to $\mathcal{V}ar(T_1, T_2)$ because parallel composition may make the domain of a resulting substitution in $Suc(\overset{li}{\rightsquigarrow}_{\mathcal{R}}, T_1) \Uparrow Suc(\overset{li}{\rightsquigarrow}_{\mathcal{R}}, T_2)$ include a variable that does not appear in $T_1 \& T_2$. Theorem 12 enables us to, given $T_1 \& T_2$, compute $Suc(\overset{li}{\rightsquigarrow}_{\mathcal{R}}, T_1)$ and $Suc(\overset{li}{\rightsquigarrow}_{\mathcal{R}}, T_2)$ separately, so-called *splitting*, instead of computing $Suc(\overset{li}{\rightsquigarrow}_{\mathcal{R}}, T_1 \& T_2)$, and then apply parallel composition to them under the variable restriction to $\mathcal{V}ar(T_1, T_2)$.

Let T be an equational term, p a position of T such that the root symbol of $T|_p$ is none of $\twoheadrightarrow$, $\top$, and $\&$. A *flattening* of T w.r.t. p is given by $T[x]_p \& (T|_p \twoheadrightarrow x)$ where x is a fresh variable [29]. Note that $T|_p$ may be a variable, but, to avoid any redundant replacement, we allow $T|_p$ to be a variable only if the replacement is in the process of linearizing a basic term in T. Thanks to Theorem 12, we can use flattening in computing the success set of T.

▶ **Theorem 13** (flattening [29]). *Let $\mathcal{R}$ be a constructor TRS, T a goal clause, and T' a flattening of T w.r.t. a position p of T. Then, $Suc(\overset{li}{\rightsquigarrow}_{\mathcal{R}}, T) = (Suc(\overset{li}{\rightsquigarrow}_{\mathcal{R}}, T'))|_{\mathcal{V}ar(T)}$ up to variable renaming.*

As in Theorem 12, we restrict $Suc(\overset{li}{\rightsquigarrow}_{\mathcal{R}}, T')$ to $\mathcal{V}ar(T)$ in Theorem 13 because a variable in T' but not in T may appear in the domain of a substitution in $Suc(\overset{li}{\rightsquigarrow}_{\mathcal{R}}, T')$, but not in T.

Thanks to Theorems 12 and 13, we can show that innermost-narrowing steps to a ground normal form in $\mathcal{T}(\{\top, \&\})$ can be replaced by leftmost ones.

▶ **Theorem 14** ([29]). *Let $\mathcal{R}$ be a constructor TRS and T a goal clause. Then, $Suc(\overset{i}{\rightsquigarrow}_{\mathcal{R}}, T) = Suc(\overset{li}{\rightsquigarrow}_{\mathcal{R}}, T)$ up to variable renaming.*

Thanks to Theorem 14, both Theorems 12 and 13 hold for $\overset{i}{\rightsquigarrow}_{\mathcal{R}}$. To make the proof of Theorem 13 simpler, we adopt $T[x]_p \& T|_p \twoheadrightarrow x$ as a result of flattening. However, thanks to Theorem 14, we may adopt $T|_p \twoheadrightarrow x \& T[x]_p$ as a result of flattening.

As mentioned above, in [29], $\mathcal{R}$ is restricted to a sufficiently complete constructor TRS without extra variables. However, sufficient completeness is not used for proving Theorems 12, 13, and 14, and the existence of extra variables does not affect the proofs of Theorems 12, 13, and 14. For this reason, Theorems 12, 13, and 14 hold for constructor TRSs with extra variables.

5 Grammar Representation for Sets of Idempotent Substitutions

In this section, we formalize grammar representations that define sets of idempotent substitutions. Since substitutions derived by narrowing steps are assumed to be idempotent, we

only deal with idempotent substitutions which introduce only *fresh* variables not appearing in any previous term. The formalization here is based on *success set equations* in [29].

In the following, a renaming δ is used to rename a particular term t and we assume that $\delta|_{\mathcal{V}ar(t)}$ is injective on $\mathcal{V}ar(t)$. For this reason, as described in Footnote 4, we write $\delta|_{\mathcal{V}ar(t)}$ instead of δ, and call $\delta|_{\mathcal{V}ar(t)}$ a *renaming for t* (simply, a renaming).

We first introduce terms to represent idempotent substitutions computed using $\cdot$ and $\Uparrow$. We prepare the signature Σ consisting of the following symbols:

- idempotent substitutions which are considered constants,   (basic elements)
- a constant $\varnothing$,   (the empty set/non-existence)
- an associative binary symbol $\bullet$,   (standard composition)
- an associative binary symbol $\&$, and   (parallel composition)
- a binary symbol REC.   (recursion with renaming)

We use infix notation for $\bullet$ and $\&$, and may omit parentheses with the precedence such that $\bullet$ has a higher priority than $\&$.

We deal with terms over Σ and some constants which are used as non-terminals of grammar representations, where we allow such constants to only appear in the first argument of REC. Note that a term without any constant may appear in the first argument of REC. Given a finite set $\mathcal{N}$ of constants, we denote the set of such terms by $\mathcal{T}(\Sigma \cup \mathcal{N})$. We assume that each constant in $\mathcal{N}$ has a term t (possibly a goal clause) as subscript such as Γ_t. For an expression $\mathrm{REC}(\Gamma_t, \delta)$, the role of Γ_t is recursion to generate terms in $\mathcal{T}(\Sigma)$. To reuse substitutions generated by recursion, we connect them with other substitutions via some renaming δ. For this reason, we restrict the second argument of REC to renamings and we require each term $\mathrm{REC}(\Gamma_t, \delta)$ to satisfy $\mathcal{VR}an(\delta) = \mathcal{V}ar(t)$.

▶ **Example 15.** The following are instances of terms in $\mathcal{T}(\Sigma)$: $\{y \mapsto \mathsf{0}\} \bullet \{x \mapsto \mathsf{s}(y)\}$, $(\{x' \mapsto \mathsf{s}(y)\} \bullet \{x \mapsto x'\}) \& \{x \mapsto \mathsf{s}(\mathsf{s}(z))\}$, $(\varnothing \& \{y \mapsto z\}) \bullet \{x \mapsto \mathsf{s}(y)\}$, and $\mathrm{REC}(\{x \mapsto \mathsf{0}, y \mapsto \mathsf{s}(y')\}, \{x' \mapsto x, y' \mapsto y\}) \bullet \{y \mapsto \mathsf{s}(x')\}$.

As described in Section 3, in computing $\sigma_1 \Uparrow \sigma_2$ from two narrowing derivations $S_1 \overset{i}{\rightsquigarrow}{}^*_{\sigma_1, \mathcal{R}} T_1$ and $S_2 \overset{i}{\rightsquigarrow}{}^*_{\sigma_2, \mathcal{R}} T_2$, we assume that $\mathcal{VR}an(\sigma_1) \cap \mathcal{VR}an(\sigma_2) = \emptyset$. To satisfy this assumption explicitly in the semantics for $\mathcal{T}(\Sigma)$, we introduce an operation $fresh_\delta(\cdot)$ of substitutions to make a substitution introduce only variables that do not appear in $\mathcal{D}om(\delta) \cup \mathcal{VR}an(\delta)$: for substitutions σ, δ, we define $fresh_\delta(\sigma)$ by $(\xi \cdot \sigma)|_{\mathcal{D}om(\sigma)}$ where ξ is a renaming such that $\mathcal{D}om(\xi) \supseteq \mathcal{VR}an(\sigma)$ and $\mathcal{VR}an(\xi|_{\mathcal{VR}an(\sigma)}) \cap (\mathcal{D}om(\delta) \cup \mathcal{VR}an(\delta) \cup \mathcal{D}om(\sigma)) = \emptyset$. The subscript δ of $fresh_\delta(\cdot)$ is used to specify freshness of variables. We say that a variable x is *fresh* w.r.t. a set X of variables if $x \notin X$.

The semantics of terms in $\mathcal{T}(\Sigma)$ to define substitutions is inductively defined as follows:

- $[\![\theta]\!] = \theta$ if θ is a substitution,
- $[\![e_1 \bullet e_2]\!] = [\![e_1]\!] \cdot [\![e_2]\!]$ if $[\![e_2]\!] \neq fail$ and $[\![e_1]\!] \neq fail$,
- $[\![e_1 \& e_2]\!] = (\theta_1 \Uparrow \theta_2)|_{\mathcal{D}om(\theta_1) \cup \mathcal{D}om(\theta_2)}$ if $[\![e_1]\!] \neq fail$ and $[\![e_2]\!] \neq fail$, where $\theta_1 = [\![e_1]\!]$ and $\theta_2 = fresh_{\theta_1}([\![e_2]\!])$,
- $[\![\mathrm{REC}(e, \delta)]\!] = (fresh_\delta([\![e]\!]) \cdot \delta)|_{\mathcal{D}om(\delta)}$ if $[\![e]\!] \neq fail$ and $\mathcal{VR}an(\delta) = \mathcal{D}om([\![e]\!])$,
- otherwise, $[\![e]\!] = fail$ (e.g., $[\![\varnothing]\!] = fail$).

Notice that a constant Γ_t is not included in $\mathcal{T}(\Sigma)$, and thus, $[\![\Gamma_t]\!]$ is not defined above. Since $\Uparrow$ may fail, we allow to have *fail*, e.g., $[\![\{y \mapsto \mathsf{s}(z)\} \bullet \{x \mapsto y\} \& \{x \mapsto \mathsf{0}\}]\!] = fail$. The number of variables appearing in a regular tree grammar defined below is finite. However, we would like to use regular tree grammars to define infinitely many substitutions such that the maximum number of variables we need cannot be fixed. To solve this problem, in the definition of $[\![\mathrm{REC}(e, \delta)]\!]$, we introduced the operation $fresh_\delta(\cdot)$ that make all variables

introduced by $[\![\,e\,]\!]$ *fresh w.r.t.* $\mathcal{D}om(\delta) \cup \mathcal{VR}an(\delta)$. In [29], this operation is implicitly considered, but in this paper, we explicitly introduced REC to the syntax in order to interpret terms in $\mathcal{T}(\Sigma)$ precisely. To assume $\mathcal{VR}an([\![\,e_1\,]\!]) \cap \mathcal{VR}an([\![\,e_2\,]\!]) = \emptyset$ for $[\![\,e_1 \,\&\, e_2\,]\!]$, we also introduced $fresh_{\theta_1}(\cdot)$ in the case of $[\![\,e_1 \,\&\, e_2\,]\!]$.

▶ **Example 16.** The expressions in Example 15 are interpreted as follows: $[\![\,\{y \mapsto 0\} \bullet \{x \mapsto \mathsf{s}(y)\}\,]\!] = \{x \mapsto \mathsf{s}(0),\, y \mapsto 0\}$, $[\![\,(\{x' \mapsto \mathsf{s}(y)\} \bullet \{x \mapsto x'\}) \,\&\, \{x \mapsto \mathsf{s}(\mathsf{s}(z))\}\,]\!] = \{x \mapsto \mathsf{s}(\mathsf{s}(z)),\, x' \mapsto \mathsf{s}(\mathsf{s}(z))\}$, $[\![\,(\varnothing \,\&\, \{y \mapsto z\}) \bullet \{x \mapsto \mathsf{s}(y)\}\,]\!] = fail$, and $[\![\,\mathrm{REC}(\{x \mapsto 0,\, y \mapsto \mathsf{s}(y')\}, \{x' \mapsto x,\, y' \mapsto y\}) \bullet \{y \mapsto \mathsf{s}(x')\}\,]\!] = \{x' \mapsto 0,\, y' \mapsto \mathsf{s}(y''),\, y \mapsto \mathsf{s}(0)\}$.

To define sets of idempotent substitutions, we adopt regular tree grammars. In the following, we drop the third component from grammars constructed below because the third one is fixed to Σ and a finite number of substitutions that are clear from production rules. A *substitution-set grammar* (SSG) for a term t_0 is a regular tree grammar $\mathcal{G} = (\Gamma_{t_0}, \mathcal{N}, \mathcal{P})$ such that $\mathcal{N}$ is a finite set of non-terminals Γ_t, $\Gamma_{t_0} \in \mathcal{N}$, and $\mathcal{P}$ is a finite set of production rules of the form $\Gamma_t \to \beta$ with $\beta \in \mathcal{T}(\Sigma \cup \mathcal{N})$. Note that $L(\mathcal{G}, \Gamma_t) = \{e \in \mathcal{T}(\Sigma) \mid \Gamma_t \to_{\mathcal{G}}^* e\}$ for each $\Gamma_t \in \mathcal{N}$. The set of substitutions defined by $\mathcal{G}$ from $\Gamma_t \in \mathcal{N}$ is defined as $[\![\,L(\mathcal{G}, \Gamma_t)\,]\!] = \{[\![\,e\,]\!] \mid e \in L(\mathcal{G}, \Gamma_t),\, [\![\,e\,]\!] \neq fail\}$.

▶ **Example 17.** The SSG $\mathcal{G}_3 = (\Gamma_x, \{\Gamma_x, \Gamma_y\}, \{\Gamma_x \to \{x \mapsto 0\} \mid \mathrm{REC}(\Gamma_y, \{x' \mapsto y\}) \bullet \{x \mapsto \mathsf{s}(x')\},\ \Gamma_y \to \mathrm{REC}(\Gamma_x, \{x' \mapsto x\}) \bullet \{y \mapsto \mathsf{s}(x')\}\})$ generates a set of expressions to define substitutions replacing x by even numbers over $0/0$ and $\mathsf{s}/1$. We have that $L(\mathcal{G}_3) = L(\mathcal{G}_3, \Gamma_x) = \{\{x \mapsto 0\},\ \mathrm{REC}((\mathrm{REC}(\{x \mapsto 0\}, \{x' \mapsto x\}) \bullet \{y \mapsto \mathsf{s}(x')\}), \{x' \mapsto y\}) \bullet \{x \mapsto \mathsf{s}(x')\},\ \ldots\}$, and $[\![\,L(\mathcal{G}_3, \Gamma_x)\,]\!] = \{\{x \mapsto \mathsf{s}^{2n}(0)\} \mid n \geq 0\}$.

6 Construction of Grammar Representations of Narrowing Trees

In this section, given a pc-CTRS and a goal clause, we show a construction of an SSG for the success set of the goal clause w.r.t. innermost narrowing of the CTRS. Since every constructor SDCTRS can be converted to an equivalent pc-CTRS w.r.t. $\xrightarrow{c}$ and $\overset{i}{\rightsquigarrow}$, we only consider pc-CTRSs. We employ the idea of narrowing trees, but we directly construct SSGs. In the following, we let $\mathcal{R}$ be a pc-CTRS over a signature $\mathcal{F}$ unless noted otherwise.

For a goal clause $T = (s_1 \twoheadrightarrow t_1) \,\&\, \cdots \,\&\, (s_n \twoheadrightarrow t_n)$, we denote the set of ground constructor terms appearing as right-hand sides of goals in T by $Crhs(T)$: $Crhs(T) = \{t_1, \ldots, t_n\} \cap \mathcal{T}(\mathcal{C}_{\mathcal{R}})$. We abuse $Crhs$ for $\mathcal{R}$ and a goal clause T: $Crhs(\mathcal{R}, T) = Crhs(T) \cup \bigcup_{\ell \to r \Leftarrow C \in \mathcal{R}} Crhs(C)$. For example, $Crhs(\mathcal{R}_1, \mathsf{e}(x) \twoheadrightarrow \mathsf{true} \,\&\, \mathsf{o}(x) \twoheadrightarrow \mathsf{true}) = \{\mathsf{true}\}$. It is clear that $Crhs(\mathcal{R}, T)$ is finite.

Let T be a goal clause that does not contain $\top$. We prepare the set of constants $\mathcal{N}_{\mathcal{R}, T} = \{\Gamma_T\} \cup \{\Gamma_{f(x_1, \ldots, x_n) \twoheadrightarrow u} \mid f/n \in \mathcal{D}_{\mathcal{R}},\ x_1, \ldots, x_n \in \mathcal{V},\ f(x_1, \ldots, x_n) \text{ is linear},\ u \in Crhs(\mathcal{R}, T) \cup (\mathcal{V} \setminus \{x_1, \ldots, x_n\})\}$. Note that $\mathcal{N}_{\mathcal{R}, T}$ is finite up to variable renaming w.r.t. subscripts, and thus, we consider $\mathcal{N}_{\mathcal{R}, T}$ a set of representatives: $\mathcal{V}ar(T') \cap \mathcal{V}ar(T'') = \emptyset$ and T' is not a variant of T'' for any different non-terminals $\Gamma_{T'}, \Gamma_{T''} \in \mathcal{N}_{\mathcal{R}, T}$. We construct an SSG from $\mathcal{R}$ and T as follows: $SSG(\mathcal{R}, T) = (\Gamma_T, \mathcal{N}_{\mathcal{R}, T}, \{\Gamma_{T'} \to \Phi_0(T') \mid \Gamma_{T'} \in \mathcal{N}_{\mathcal{R}, T}\})$, where $\Phi_b(\cdot)$ with $b \in \{0, 1\}$ is inductively defined as follows:

Splitting $\Phi_b(T_1 \,\&\, \cdots \,\&\, T_n) = \Phi_1(T_1) \,\&\, \cdots \,\&\, \Phi_1(T_n)$,

Narrowing $\Phi_0(f(x_1, \ldots, x_n) \twoheadrightarrow u) = \Phi_1(T_1) \bullet \sigma_1 \mid \cdots \mid \Phi_1(T_m) \bullet \sigma_m$ if $f(x_1, \ldots, x_n)$ is basic and linear, and $x_1, \ldots, x_n \in \mathcal{V}$, where $\{(T', \sigma) \mid (f(x_1, \ldots, x_n) \twoheadrightarrow u) \overset{i}{\rightsquigarrow}_{\sigma, \mathcal{R}} T',\ \mathcal{VR}an(\sigma_i) \cap (\bigcup_{\Gamma_{T'} \in \mathcal{N}_{\mathcal{R}, T}} \mathcal{V}ar(T')) = \emptyset\} = \{(T_1, \sigma_1), \ldots, (T_m, \sigma_m)\}$,

Narrowing $\Phi_b(t \twoheadrightarrow u) = mgu(\{t \approx u\})$ if $t, u \in \mathcal{T}(\mathcal{C}_{\mathcal{R}}, \mathcal{V})$ and t, u are unifiable,

Failure $\Phi_b(t \twoheadrightarrow u) = \varnothing$ if $t, u \in \mathcal{T}(\mathcal{C}_{\mathcal{R}}, \mathcal{V})$ and t, u are not unifiable,

Recursion $\Phi_1(f(x_1, \ldots, x_n) \twoheadrightarrow u) = \text{REC}(\Gamma_{f(x'_1, \ldots, x'_n) \twoheadrightarrow u'}, \{x_1 \mapsto x'_1, \ldots, x_n \mapsto x'_n\} \cup \delta)$ if $f(x_1, \ldots, x_n)$ is basic and linear, $\Gamma_{f(x'_1, \ldots, x'_n) \twoheadrightarrow u'} \in \mathcal{N}_{\mathcal{R},T}$, $x_1, \ldots, x_n \in \mathcal{V} \setminus \{x'_1, \ldots, x'_n\}$, $u' \in \mathcal{T}(\mathcal{C}_{\mathcal{R}}) \cup (\mathcal{V} \setminus (\{x_1, \ldots, x_n\} \cup \mathcal{V}ar(u)))$, and either $u = u' \in \mathcal{T}(\mathcal{C}_{\mathcal{R}})$ or $u, u' \in \mathcal{V}$, where if $u \in \mathcal{T}(\mathcal{C}_{\mathcal{R}})$, then $\delta = id$, and otherwise, $\delta = \{u \mapsto u'\}$), and

Flattening $\Phi_b(f(u_1, \ldots, u_n) \twoheadrightarrow u) = \Phi_1(f(y_1, \ldots, y_n) \twoheadrightarrow y) \mathrel{\&} (u_1 \twoheadrightarrow y_1) \mathrel{\&} \cdots \mathrel{\&} (u_n \twoheadrightarrow y_n) \mathrel{\&} (u \twoheadrightarrow y)$ if $f(u_1, \ldots, u_n) \twoheadrightarrow u$ is not a variant of $f(x'_1, \ldots, x'_n) \twoheadrightarrow u'$ with $\mathcal{V}ar(u_1, \ldots, u_n, u) \cap (\{x'_1, \ldots, x'_n\} \cup \mathcal{V}ar(u')) = \emptyset$ for any $\Gamma_{f(x'_1, \ldots, x'_n) \twoheadrightarrow u'} \in \mathcal{N}_{\mathcal{R},T}$, where $y_1, \ldots, y_n$, are fresh distinct variables w.r.t. $\mathcal{V}ar(u_1, \ldots, u_n, u) \cup \bigcup_{\Gamma_{T'} \in \mathcal{N}_{\mathcal{R},T}} \mathcal{V}ar(T')$. Note that we do not have to added the goal $u \twoheadrightarrow y$ to the result if $u \in Crhs(\mathcal{R}, T)$.

Note that we may omit $\Gamma_{T'}$ and its production rules if $\Gamma_{T'}$ is not relevant to Γ_T. The subscript b of $\Phi_b(\cdot)$ is used to specify whether the call of $\Phi_b(\cdot)$ is initial or not. Without the subscript, for $\Gamma_{f(x_1, \ldots, x_n) \twoheadrightarrow u}$, we only construct $\Gamma_{f(x_1, \ldots, x_n) \twoheadrightarrow u} \to \text{REC}(id, \Gamma_{f(x_1, \ldots, x_n) \twoheadrightarrow u})$ which is meaningless. The definition of $\Phi_b(\cdot)$ follows the definition of a single step of narrowing, splitting under parallel composition, and flattening in the natural way. For example, the semantics of REC takes renamings for $\mathcal{V}ar(\ell, r, C) \cap \mathcal{V}ar(S) = \emptyset$ in the definition of innermost-narrowing into account and enables us to reuse substitutions generated by the first argument of REC.

▶ **Example 18.** For $\mathcal{R}_1$ and the goal clause $\mathsf{e}(x) \twoheadrightarrow \mathsf{true} \mathrel{\&} \mathsf{o}(x) \twoheadrightarrow \mathsf{true}$, we prepare constants $\Gamma_{\mathsf{e}(x) \twoheadrightarrow \mathsf{true} \& \mathsf{o}(x) \twoheadrightarrow \mathsf{true}}$, $\Gamma_{\mathsf{e}(x') \twoheadrightarrow \mathsf{true}}$, and $\Gamma_{\mathsf{o}(x'') \twoheadrightarrow \mathsf{true}}$ because we have that $Crhs(\mathcal{R}_1, \mathsf{e}(x) \twoheadrightarrow \mathsf{true} \mathrel{\&} \mathsf{o}(x) \twoheadrightarrow \mathsf{true}) = \{\mathsf{true}\}$. For the goal $\mathsf{e}(x') \twoheadrightarrow \mathsf{true}$, we have the following conversion:

$$\Phi_0(\mathsf{e}(x') \twoheadrightarrow \mathsf{true}) = id \bullet \{x' \mapsto 0\} \mid \big(\text{REC}(\Gamma_{\mathsf{o}(x'') \twoheadrightarrow \mathsf{true}}, \{x_1 \mapsto x''\}) \mathrel{\&} id\big) \bullet \{x' \mapsto \mathsf{s}(x_1)\}$$
$$\mid \big(\text{REC}(\Gamma_{\mathsf{e}(x') \twoheadrightarrow \mathsf{true}}, \{x_2 \mapsto x'\}) \mathrel{\&} \varnothing\big) \bullet \{x' \mapsto \mathsf{s}(x_2)\}$$

From the conversion above, the SSG $\mathcal{G}_2$ with the following production rules is constructed:

$$\Gamma_{\mathsf{e}(x) \twoheadrightarrow \mathsf{true} \& \mathsf{o}(x) \twoheadrightarrow \mathsf{true}} \to \text{REC}(\Gamma_{\mathsf{e}(x') \twoheadrightarrow \mathsf{true}}, \{x \mapsto x'\}) \mathrel{\&} \text{REC}(\Gamma_{\mathsf{o}(x'') \twoheadrightarrow \mathsf{true}}, \{x \mapsto x''\})$$
$$\Gamma_{\mathsf{e}(x') \twoheadrightarrow \mathsf{true}} \to id \bullet \{x' \mapsto 0\} \mid \big(\text{REC}(\Gamma_{\mathsf{o}(x'') \twoheadrightarrow \mathsf{true}}, \{x_1 \mapsto x''\}) \mathrel{\&} id\big) \bullet \{x' \mapsto \mathsf{s}(x_1)\}$$
$$\mid \big(\text{REC}(\Gamma_{\mathsf{e}(x') \twoheadrightarrow \mathsf{true}}, \{x_2 \mapsto x'\}) \mathrel{\&} \varnothing\big) \bullet \{x' \mapsto \mathsf{s}(x_2)\}$$
$$\Gamma_{\mathsf{o}(x'') \twoheadrightarrow \mathsf{true}} \to \varnothing \bullet \{x'' \mapsto 0\} \mid \big(\text{REC}(\Gamma_{\mathsf{e}(x') \twoheadrightarrow \mathsf{true}}, \{x_3 \mapsto x'\}) \mathrel{\&} id\big) \bullet \{x'' \mapsto \mathsf{s}(x_3)\}$$
$$\mid \big(\text{REC}(\Gamma_{\mathsf{o}(x'') \twoheadrightarrow \mathsf{true}}, \{x_4 \mapsto x''\}) \mathrel{\&} \varnothing\big) \bullet \{x'' \mapsto \mathsf{s}(x_4)\}$$

▶ **Theorem 19.** *Let T be a goal clause without $\top$. Then, $\llbracket L(SSG(\mathcal{R}, T), \Gamma_T) \rrbracket = Suc(\overset{i}{\leadsto}_{\mathcal{R}}, T)$ up to variable renaming.*

Note that Theorem 19 corresponds to [29, Theorem 20]. For a constructor SDCTRS $\mathcal{R}$ and a goal clause T, Theorem 6 enables us to use $SSG(Pc(\mathcal{R}), T)$ for $\mathcal{R}$.

7 Simplification of Grammar Representations

In this section, we show some methods to simplify production rules of SSGs. Given an SSG $\mathcal{G} = (\Gamma_T, \mathcal{N}, \mathcal{P})$, we extend the semantics of terms in $\mathcal{T}(\Sigma)$ to sets of terms in $\mathcal{T}(\Sigma \cup \mathcal{N})$ as follows: $\{\!| e |\!\}_{\mathcal{G}} = \llbracket L((\Gamma_T, \mathcal{N} \cup \{\Gamma_e\}, \mathcal{P} \cup \{\Gamma_e \to e\}), \Gamma_e) \rrbracket$ for $e \in \mathcal{T}(\Sigma \cup \mathcal{N})$, where $\Gamma_e \notin \mathcal{N}$. We say that terms $e_1, e_2 \in \mathcal{T}(\Sigma \cup \mathcal{N})$ are *semantically equivalent w.r.t.* $\mathcal{G}$ if $\{\!| e_1 |\!\}_{\mathcal{G}} = \{\!| e_2 |\!\}_{\mathcal{G}}$ up to variable renaming.

We first compute subexpressions consisting of substitutions, $\bullet$, &, and $\varnothing$. The following equivalences trivially hold:

▶ **Theorem 20.** *Let $\mathcal{G} = (\Gamma_T, \mathcal{N}, \mathcal{P})$, θ_1, θ_2 idempotent substitutions, and $e \in \mathcal{T}(\Sigma \cup \mathcal{N})$. Then, all of the following hold: $\{\!| \theta_1 \bullet \theta_2 |\!\}_{\mathcal{G}} = \{\!| \theta_1 \cdot \theta_2 |\!\}_{\mathcal{G}}$, $\{\!| e \bullet \varnothing |\!\}_{\mathcal{G}} = \{\!| \varnothing \bullet e |\!\}_{\mathcal{G}} = \{\!| \varnothing \mathrel{\&} e |\!\}_{\mathcal{G}} = \{\!| e \mathrel{\&} \varnothing |\!\}_{\mathcal{G}} = \{\!| \varnothing |\!\}_{\mathcal{G}}$ $(= \emptyset)$, and $\{\!| id \mathrel{\&} e |\!\}_{\mathcal{G}} = \{\!| e \mathrel{\&} id |\!\}_{\mathcal{G}} = \{\!| e |\!\}_{\mathcal{G}}$.*

Following Theorem 20, we simplify subexpressions to the smallest one among semantically equivalent terms w.r.t. $\mathcal{G}$ (e.g., replace $e \bullet \varnothing$ by $\varnothing$) as much as possible.

▶ **Example 21.** The production rules of $\mathcal{G}_2$ in Example 18 are simplified as follows:

$$\Gamma_{\mathsf{e}(x)\twoheadrightarrow\mathsf{true}\&\mathsf{o}(x)\twoheadrightarrow\mathsf{true}} \to \mathrm{REC}(\Gamma_{\mathsf{e}(x')\twoheadrightarrow\mathsf{true}}, \{x \mapsto x'\}) \,\&\, \mathrm{REC}(\Gamma_{\mathsf{o}(x'')\twoheadrightarrow\mathsf{true}}, \{x \mapsto x''\})$$

$$\Gamma_{\mathsf{e}(x')\twoheadrightarrow\mathsf{true}} \to \{x' \mapsto 0\} \mid \mathrm{REC}(\Gamma_{\mathsf{o}(x'')\twoheadrightarrow\mathsf{true}}, \{x_1 \mapsto x''\}) \bullet \{x' \mapsto \mathsf{s}(x_1)\}$$

$$\Gamma_{\mathsf{o}(x'')\twoheadrightarrow\mathsf{true}} \to \mathrm{REC}(\Gamma_{\mathsf{e}(x')\twoheadrightarrow\mathsf{true}}, \{x_3 \mapsto x'\}) \bullet \{x'' \mapsto \mathsf{s}(x_3)\}$$

The occurrence of & in SSGs makes it difficult to simplify and analyze grammar representations of narrowing trees. Since the second and third production rules in Example 21 no longer contain &, we focus on $\mathrm{REC}(\Gamma_{\mathsf{e}(x')\twoheadrightarrow\mathsf{true}}, \{x \mapsto x'\}) \,\&\, \mathrm{REC}(\Gamma_{\mathsf{o}(x'')\twoheadrightarrow\mathsf{true}}, \{x \mapsto x''\})$ which is the right-hand side of the first rule. Let us consider the sets L_1, L_2 of terms substituted for x by means of substitutions in $\{\!|\, \mathrm{REC}(\Gamma_{\mathsf{e}(x')\twoheadrightarrow\mathsf{true}}, \{x \mapsto x'\}) \,|\!\}_{\mathcal{G}_2}$ and $\{\!|\, \mathrm{REC}(\Gamma_{\mathsf{o}(x'')\twoheadrightarrow\mathsf{true}}, \{x \mapsto x''\}) \,|\!\}_{\mathcal{G}_2}$, respectively: $L_1 = \{\sigma x \mid \sigma \in \{\!|\, \mathrm{REC}(\Gamma_{\mathsf{e}(x')\twoheadrightarrow\mathsf{true}}, \{x \mapsto x'\}) \,|\!\}_{\mathcal{G}_2}\}$, and $L_2 = \{\sigma x \mid \sigma \in \{\!|\, \mathrm{REC}(\Gamma_{\mathsf{o}(x'')\twoheadrightarrow\mathsf{true}}, \{x \mapsto x''\}) \,|\!\}_{\mathcal{G}_2}\}$. If $L_1 \cap L_2 = \emptyset$, then $\{\!|\, \mathrm{REC}(\Gamma_{\mathsf{e}(x')\twoheadrightarrow\mathsf{true}}, \{x \mapsto x'\}) \,\&\, \mathrm{REC}(\Gamma_{\mathsf{o}(x'')\twoheadrightarrow\mathsf{true}}, \{x \mapsto x''\}) \,|\!\}_{\mathcal{G}_2} = \emptyset$ and we obtain $\Gamma_{\mathsf{e}(x)\twoheadrightarrow\mathsf{true}\&\mathsf{o}(x)\twoheadrightarrow\mathsf{true}} \to \varnothing$ which is our goal of simplification in this section. To generate L_1 and L_2, we transform the second and third production rules in Example 21 into a regular tree grammar generating the sets L_1 and L_2. In the rest of this section, we assume that the signature $\mathcal{F}$ contains a constant.

Let $\mathcal{G}$ be an SSG $(\Gamma_{T_0}, \mathcal{N}, \mathcal{P})$ and T a goal clause such that $\Gamma_T \in \mathcal{N}$. We denote by $\mathcal{P}|_{\Gamma_T}$ the set of production rules that are reachable from Γ_T. We assume that any rule in $\mathcal{P}|_{\Gamma_T}$ is of the form $\Gamma_{T'} \to \theta_1 \mid \cdots \mid \theta_m \mid \mathrm{REC}(\Gamma_{T_1}, \delta_1) \bullet \sigma_1 \mid \cdots \mid \mathrm{REC}(\Gamma_{T_n}, \delta_n) \bullet \sigma_n$, where $\theta_1, \ldots, \theta_m, \sigma_1, \ldots, \sigma_n$ are idempotent substitutions. Note that $\Gamma_{T'} \to \mathrm{REC}(\Gamma_{T''}, \delta)$ is considered $\Gamma_{T'} \to \mathrm{REC}(\Gamma_{T''}, \delta) \bullet id$. Note also that the following construction is applicable under this assumption. The regular tree grammar obtained from $\mathcal{G}$ and a variable x in T, written as $RTG(\mathcal{G}, T, x)$, is $(\Gamma_T^x, \mathcal{N}', \mathcal{P}' \cup \mathcal{P}'' \cup \{A \to g(\overbrace{A, \ldots, A}^{n}) \mid g/n \in \mathcal{C}_{\mathcal{R}}\})$ such that

- $\mathcal{N}' = \{\Gamma_{T'}^{x'} \mid \Gamma_{T'} \in \mathcal{N}, \ x' \in \mathcal{V}ar(T')\} \cup \{A\}$, and
- $\mathcal{P}' = \{\Gamma_{T'}^{x'} \to \xi_{\mathcal{V}ar(\theta_i x')}(\theta_i x') \mid x' \in \mathcal{V}ar(T'), \ \Gamma_{T'} \to \theta_i \in \mathcal{P}\}$, and
- $\mathcal{P}'' = \{\Gamma_{T'}^{x'} \to \left(\{y \mapsto \Gamma_{T_j}^{\delta_j y} \mid y \in \mathcal{D}om(\delta_j)\} \cup \xi_{\mathcal{V}ar(\sigma_j x')\backslash\mathcal{D}om(\delta_j)}\right)(\sigma_j x') \mid x' \in \mathcal{V}ar(T'),$
  $\Gamma_{T'} \to \mathrm{REC}(\Gamma_{T_j}, \delta_j) \bullet \sigma_j \in \mathcal{P}\}$,

where $\xi_X = \{y \mapsto A \mid y \in X\}$, which corresponds to ξ in the definition of $fresh_\delta(\cdot)$. Note that the non-terminal A generates $\mathcal{T}(\mathcal{C}_{\mathcal{R}})$ and corresponds to a fresh variable.

▶ **Theorem 22.** *Let $\mathcal{G}$ be an SSG $(\Gamma_{T_0}, \mathcal{N}, \mathcal{P})$, $\Gamma_{T_1}, \Gamma_{T_2} \in \mathcal{N}$, $x \in \mathcal{V}$, $x_1 \in \mathcal{V}ar(T_1)$, $x_2 \in \mathcal{V}ar(T_2)$, $RTG(\mathcal{G}, T_1, x_1), RTG(\mathcal{G}, T_2, x_2)$ be constructed, and δ_1, δ_2 be renamings such that $\mathcal{V}\mathcal{R}an(\delta_i) = \mathcal{V}ar(T_i)$ and $\delta_i x = x_i$ for $i = 1, 2$. If $L(RTG(\mathcal{G}, T_1, x_1)) \cap L(RTG(\mathcal{G}, T_2, x_2)) = \emptyset$, then $\{\!|\, \mathrm{REC}(\Gamma_{T_1}, \delta_1) \,\&\, \mathrm{REC}(\Gamma_{T_2}, \delta_2) \,|\!\}_{\mathcal{G}} = \{\!|\, \varnothing \,|\!\}_{\mathcal{G}}$.*

▶ **Example 23.** From the production rules in Example 21, we obtain the following regular tree grammars:

- $\mathcal{G}_2' = RTG(\mathcal{G}_2, \mathsf{e}(x') \twoheadrightarrow \mathsf{true}, x') = (\Gamma_{\mathsf{e}(x')\twoheadrightarrow\mathsf{true}}^{x'}, \{\Gamma_{\mathsf{e}(x')\twoheadrightarrow\mathsf{true}}^{x'}, \Gamma_{\mathsf{o}(x'')\twoheadrightarrow\mathsf{true}}^{x''}\}, \mathcal{P}')$, and
- $\mathcal{G}_2'' = RTG(\mathcal{G}_2, \mathsf{o}(x'') \twoheadrightarrow \mathsf{true}, x'') = (\Gamma_{\mathsf{o}(x'')\twoheadrightarrow\mathsf{true}}^{x''}, \{\Gamma_{\mathsf{e}(x')\twoheadrightarrow\mathsf{true}}^{x'}, \Gamma_{\mathsf{o}(x'')\twoheadrightarrow\mathsf{true}}^{x''}\}, \mathcal{P}')$,

where

- $\mathcal{P}' = \{\ \Gamma_{\mathsf{e}(x')\twoheadrightarrow\mathsf{true}}^{x'} \to 0 \mid \mathsf{s}(\Gamma_{\mathsf{o}(x'')\twoheadrightarrow\mathsf{true}}^{x''}), \ \ \Gamma_{\mathsf{o}(x'')\twoheadrightarrow\mathsf{true}}^{x''} \to \mathsf{s}(\Gamma_{\mathsf{e}(x')\twoheadrightarrow\mathsf{true}}^{x'})\ \}$.

We can decide the intersection emptiness problem of $L(\mathcal{G}_2')$ and $L(\mathcal{G}_2'')$, and the answer is true: $L(\mathcal{G}_2') \cap L(\mathcal{G}_2'') = \emptyset$. Thanks to Theorem 22, we can replace the expression $\mathrm{REC}(\Gamma_{\mathsf{e}(x')\twoheadrightarrow\mathsf{true}}, \{x \mapsto x'\}) \,\&\, \mathrm{REC}(\Gamma_{\mathsf{o}(x'')\twoheadrightarrow\mathsf{true}}, \{x \mapsto x''\})$ by $\varnothing$, and thus we can transform the first production rule in Example 21 into $\Gamma_{\mathsf{e}(x)\twoheadrightarrow\mathsf{true}\&\mathsf{o}(x)\twoheadrightarrow\mathsf{true}} \to \varnothing$.

In summary, the simplification proposed in this section is to replace subexpressions by semantically equivalent smaller ones as much as possible by following Theorems 20 and 22. This simplification always halts because the number of Σ-symbols in equations is strictly decreasing at every simplification step. In addition, it is clear that results of the simplification are unique.

If a constructor SDCTRS $\mathcal{R}$ has a nest of defined symbols or a goal clause T contains, e.g., $(f(\vec{x}) \twoheadrightarrow x') \mathbin{\&} (g(\vec{y}) \twoheadrightarrow y')$, to simplify $SSG(\mathit{Trs}(Pc(\mathcal{R})), T)$ as much as possible, we apply the simplification based on Theorem 22 at least once, i.e., we try to solve the intersection emptiness problem of regular tree grammars at least once, which is EXPTIME-complete. The number of occurrence of $\bullet$ and $\mathbin{\&}$ is at most $O(n)$, where n is the size of $\mathcal{R}$ and T. Therefore, the cost of the overall simplification is EXPTIME-complete.

8   Applications

In this section, we show that grammar representations of narrowing trees are useful to prove (1) infeasibility of conditional critical pairs of $\mathcal{R}_1$ and (2) quasi-reducibility of $\mathcal{R}_1$ with usual sorts for the non-negative integers and the boolean values.

Two conditional rewrite rules $\ell_1 \to r_1 \Leftarrow C_1$ and $\ell_2 \to r_2 \Leftarrow C_2$ that are renamed to have no shared variable are said to be *overlapping* if there exists a non-variable position p of ℓ_1 such that $\ell_1|_p$ and ℓ_2 are unifiable, and $p \neq \varepsilon$ if one of the rules is a renamed variant of the other. In this case, given $\sigma = mgu(\{\ell_1|_p \approx \ell_2\})$, the triple $(\sigma(\ell_1[r_2]_p), \sigma r_1, \sigma C_1 \mathbin{\&} \sigma C_2)$, denoted by $\langle \sigma(\ell_1[r_2]_p), \sigma r_1 \rangle \Leftarrow \sigma C_1 \mathbin{\&} \sigma C_2$, is called a *conditional critical pair* of $\mathcal{R}$. A conditional critical pair $\langle s, t \rangle \Leftarrow s_1 \twoheadrightarrow t_1 \mathbin{\&} \cdots \mathbin{\&} s_k \twoheadrightarrow t_k$ is called *infeasible* if there exists no substitution θ such that $\theta s_i \to_{\mathcal{R}}^* \theta t_i$ for all $1 \leq i \leq k$, and called *joinable* if $\theta s \downarrow_{\mathcal{R}} \theta t$ for any substitution θ such that $\theta s_i \to_{\mathcal{R}}^* \theta t_i$ for all $1 \leq i \leq k$. Note that infeasible conditional critical pairs are joinable and unconditional critical pairs are feasible. Therefore, from [4, Theorem 3.8] and [21, Theorem 3], an operationally terminating CTRS $\mathcal{R}$ is confluent if all critical pairs of $\mathcal{R}$ are infeasible.

▶ **Example 24.** Consider $\mathcal{R}_1$ in Section 1. It follows from Example 23 that $Suc(\stackrel{i}{\rightsquigarrow}_{\mathcal{R}_1}, \mathsf{o}(x) \twoheadrightarrow \mathsf{true} \mathbin{\&} \mathsf{e}(x) \twoheadrightarrow \mathsf{true}) = \emptyset$. This means that there exists no constructor term t such that $\mathsf{o}(t) \stackrel{c}{\to}_{\mathcal{R}_1}^* \mathsf{true}$ and $\mathsf{e}(t) \stackrel{c}{\to}_{\mathcal{R}_1}^* \mathsf{true}$. Assume that there exists a term t such that $\mathsf{o}(t) \to_{\mathcal{R}_1}^* \mathsf{true}$ and $\mathsf{e}(t) \to_{\mathcal{R}_1}^* \mathsf{true}$. Since $\stackrel{c}{\to}_{\mathcal{R}_1} = \stackrel{c}{\to}_{\mathit{Trs}(\mathcal{R}_1)}$ and $\mathit{Trs}(\mathcal{R}_1)$ is non-erasing, t should be a ground constructor term. Since $\mathit{Trs}(\mathcal{R}_1)$ is a constructor system, we have that $\mathsf{o}(t) \stackrel{c}{\to}_{\mathcal{R}_1}^* \mathsf{true}$ and $\mathsf{e}(t) \stackrel{c}{\to}_{\mathcal{R}_1}^* \mathsf{true}$, and hence $\mathsf{o}(x) \stackrel{i}{\rightsquigarrow}_{\theta_1, \mathcal{R}_1}^* \mathsf{true}$ and $\mathsf{e}(x) \stackrel{i}{\rightsquigarrow}_{\theta_2, \mathcal{R}_1}^* \mathsf{true}$ for some constructor substitutions θ_1, θ_2. It follows from Theorem 12 that $\theta_1 \Uparrow \theta_2 \in Suc(\stackrel{i}{\rightsquigarrow}_{\mathcal{R}_1}, \mathsf{o}(x) \twoheadrightarrow \mathsf{true} \mathbin{\&} \mathsf{e}(x) \twoheadrightarrow \mathsf{true})$. This contradicts the fact that $Suc(\stackrel{i}{\rightsquigarrow}_{\mathcal{R}_1}, \mathsf{o}(x) \twoheadrightarrow \mathsf{true} \mathbin{\&} \mathsf{e}(x) \twoheadrightarrow \mathsf{true}) = \emptyset$. Therefore, all critical pairs of $\mathcal{R}_1$ are infeasible, and hence $\mathcal{R}_1$ is confluent.

A CTRS $\mathcal{R}$ is called *quasi-reducible* [16] if any ground basic term is not a normal form. $\mathcal{R}$ is called *sufficiently complete* if for every ground term t, there exists a ground constructor term u such that $t \to_{\mathcal{R}}^* u$ [12]. Note that if an operationally terminating CTRS is quasi-reducible, then the CTRS is sufficiently complete.

▶ **Example 25.** CTRS $\mathcal{R}_1$ in Section 1 is not quasi-reducible since $\mathsf{e}(\mathsf{true})$ is not defined. Thus, let us consider the sorts with $0 : \mathit{nat}$, $\mathsf{s} : \mathit{nat} \to \mathit{nat}$, $\mathsf{true}, \mathsf{false} : \mathit{bool}$, and $\mathsf{e}, \mathsf{o} : \mathit{nat} \to \mathit{bool}$. For quasi-reducibility of $\mathcal{R}_1$ with the sorts, it suffices to show that $\mathsf{e}(\mathsf{s}^n(0))$ and $\mathsf{o}(\mathsf{s}^n(0))$ with $n \geq 0$ are reducible. It follows from the unconditional rules $\mathsf{e}(0) \to \mathsf{true}$ and $\mathsf{o}(0) \to \mathsf{false}$ that $\mathsf{e}(0)$ and $\mathsf{o}(0)$ are reducible. From the production rules in Example 23, we can show that $L(\mathcal{G}_2') \cup L(\mathcal{G}_2'') = \mathcal{T}(\{0, \mathsf{s}\})$, and hence $\mathsf{e}(\mathsf{s}^n(0))$ and $\mathsf{o}(\mathsf{s}^n(0))$ with $n > 0$ are reducible. Therefore, $\mathcal{R}_1$ with the sorts is quasi-reducible, and hence sufficiently complete.

9 Related Work

One of the closest related work to be compared with our results must be reachability analysis for CTRSs. A well-investigated approach is *tree automata techniques* (cf. [7, 6]): given a (C)TRS and two terms s, t, we construct a tree automaton that over-approximately recognizes all descendants of any ground instance of s, and solves the intersection emptiness problem between the automaton and another one for ground instances of t. To prove infeasibility of $\mathsf{o}(x) \twoheadrightarrow \mathsf{true} \,\&\, \mathsf{e}(x) \twoheadrightarrow \mathsf{true}$ w.r.t. $\xrightarrow{c}_{\mathcal{R}_1}$ via reachability, we convert it to the reachability problem from ground instances of $\mathsf{c}(\mathsf{o}(x), \mathsf{e}(x))$ to $\mathsf{c}(\mathsf{true}, \mathsf{true})$. Tree automata techniques need overapproximation for non-linear terms, and thus, the reachability problem is solved as the reachability from $\mathsf{c}(\mathsf{o}(x'), \mathsf{e}(x''))$ to $\mathsf{c}(\mathsf{true}, \mathsf{true})$. Due to this linearization, the non-existence of a ground instance of x cannot be proved. The method in [36] for proving infeasibility of conditional critical pairs analyzes reachability using the underlying TRSs – { $\mathsf{e}(0) \to \mathsf{true}$, $\mathsf{e}(\mathsf{s}(x)) \to \mathsf{true}$, $\mathsf{e}(\mathsf{s}(x)) \to \mathsf{false}$, $\mathsf{o}(0) \to \mathsf{false}$, $\mathsf{o}(\mathsf{s}(x)) \to \mathsf{true}$, $\mathsf{o}(\mathsf{s}(x)) \to \mathsf{false}$ } for $\mathcal{R}_1$ – , and thus, the non-existence of a ground t with $\mathsf{o}(t) \to^*_{\mathcal{R}_1} \mathsf{true}$ and $\mathsf{e}(t) \to^*_{\mathcal{R}_1} \mathsf{true}$ cannot be proved. On the other hand, the approach in this paper is to construct a regular tree grammar that can be seen as a tree automaton, and that recognizes ground terms given at an argument of a defined symbol we are interested in.

Another important related work is a *semantic approach* to infeasibility analysis for conditional rewrite rules and conditional critical pairs of CTRSs [19, 18], which uses AGES [11] based on the methods in [20]. The semantic approach reduces infeasibility of conditions $s_1 \twoheadrightarrow t_1, \ldots, s_k \twoheadrightarrow t_k$ to the existence of a logical model for the theory $\overline{\mathcal{R}} \cup \{\neg(\exists \overrightarrow{X}. \ s_1 \to^* t_1 \wedge \cdots \wedge s_k \to^* t_k)\}$, where $X = \mathcal{V}ar(s_1, t_1, \ldots, s_k, t_k)$ and $\overline{\mathcal{R}}$ is a first-order theory obtained by $\mathcal{R}$. The power of proving infeasibility relies on that of generating a model for the theory. For example, infeasibility of $x < y \twoheadrightarrow \mathsf{true}, y < x \twoheadrightarrow \mathsf{true}$ w.r.t. $\mathcal{R}_4 = \{\ 0 < \mathsf{s}(y) \to \mathsf{true}, \ x < 0 \to \mathsf{false}, \ \mathsf{s}(x) < \mathsf{s}(y) \to x < y \ \}$ can be reduced to the existence of a model for $\overline{\mathcal{R}_4} \cup \{\neg(\forall x, y. \ x < y \to^* \mathsf{true} \wedge y < x \to^* \mathsf{true})\}$, but AGES did not find any model for the theory via its web interface with default parameters. The power of our method for proving infeasibility relies on the success of simplifying SSGs to $\Gamma_T \to \varnothing$. For this reason, it is not so easy to compare these two approaches from theoretical point of view to prove infeasibility of conditions. On the other hand, our result can be used to prove quasi-reducibility of $\mathcal{R}_1$ with usual sorts for the non-negative integers and the boolean values.

10 Conclusion

In this paper, we extended grammar representations of narrowing trees to constructor SDCTRSs, and showed that grammar representations are useful to prove confluence and quasi-reducibility of a normal CTRS. We will implement the construction and simplification of grammar representations for narrowing trees, and will introduce them into CO3 [25] to use them to prove confluence of constructor SDCTRSs. In addition, we will make an empirical comparison of the tree automata approach, the semantic approach, and ours to infeasibility analysis of constructor SDCTRSs after implementing our method. Narrowing trees define constructor substitutions obtained by innermost narrowing. For this reason, the usefulness is limited to constructor-based rewriting only. A further direction of this research will be to extend narrowing trees to other kinds of narrowing, e.g., *basic* narrowing [14].

References

1 Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

2 Adel Bouhoula and Florent Jacquemard. Sufficient completeness verification for conditional and constrained TRS. *Journal of Applied Logic*, 10(1):127–143, 2012.

3 Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. Tree Automata Techniques and Applications, 2007. Release October, 12th 2007.

4 Nachum Dershowitz, Mitsuhiro Okada, and G. Sivakumar. Canonical conditional rewrite systems. In *Proceedings of the 9th International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 538–549. Springer, 1988.

5 Francisco Durán, Salvador Lucas, José Meseguer, Claude Marché, and Xavier Urbain. Proving termination of membership equational programs. In Nevin Heintze and Peter Sestoft, editors, *Proceedings of the 2004 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 147–158. ACM, 2004.

6 Guillaume Feuillade and Thomas Genet. Reachability in conditional term rewriting systems. *Electronic Notes in Theoretical Computer Science*, 86(1):133–146, 2003.

7 Thomas Genet. Decidable approximations of sets of descendants and sets of normal forms. In Tobias Nipkow, editor, *Proceedings of the 9th International Conference on Rewriting Techniques and Applications*, volume 1379 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 1998.

8 Karl Gmeiner. CoScart: Confluence prover in Scala. In Ashish Tiwari and Takahito Aoto, editors, *Proceedings of the 4th International Workshop on Confluence*, page 45, 2015.

9 Karl Gmeiner and Naoki Nishida. Notes on structure-preserving transformations of conditional term rewrite systems. In Manfred Schmidt-Schauß, Masahiko Sakai, David Sabel, and Yuki Chiba, editors, *Proceedings of the first International Workshop on Rewriting Techniques for Program Transformations and Evaluation*, volume 40 of *OpenAccess Series in Informatics*, pages 3–14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2014.

10 Karl Gmeiner, Naoki Nishida, and Bernhard Gramlich. Proving confluence of conditional term rewriting systems via unravelings. In Nao Hirokawa and Vincent van Oostrom, editors, *Proceedings of the 2nd International Workshop on Confluence*, pages 35–39, 2013.

11 Raúl Gutiérrez, Salvador Lucas, and Patricio Reinoso. A tool for the automatic generation of logical models of order-sorted first-order theories. In Alicia Villanueva, editor, *Proceedings of the XVI Jornadas sobre Programación y Lenguages*, pages 215–230, 2016. Tool available at `http://zenon.dsic.upv.es/ages/`.

12 John V. Guttag. *The Specification and Application to Programming of Abstract Data Types.* PhD thesis, University of Tronto, Toronto, Canada, 1975.

13 Manuel V. Hermenegildo and Francesca Rossi. On the correctness and efficiency of independent and-parallelism in logic programs. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 369–389. MIT Press, 1989.

14 Jean-Marie Hullot. Canonical forms and unification. In Wolfgang Bibel and Robert A. Kowalski, editors, *Proceedings of the 5th Conference on Automated Deduction*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334. Springer, 1980.

15 Deepak Kapur, Paliath Narendran, and Hantao Zhang. On sufficient-completeness and related properties of term rewriting systems. *Acta Informatica*, 24(4):395–415, 1987.

16 Emmanuel Kounalis. Completeness in data type specifications. In Bob F. Caviness, editor, *Proceedings of the European Conference on Computer Algebra*, volume 204 of *Lecture Notes in Computer Science*, pages 348–362. Springer, 1985.

17   Marija Kulaš. A practical view on renaming. In Sibylle Schwarz and Janis Voigtländer, editors, *Proceedings of the 29th and 30th Workshops on (Constraint) Logic Programming and 24th International Workshop on Functional and (Constraint) Logic Programming, and 24th International Workshop on Functional and (Constraint) Logic Programming*, volume 234 of *Electronic Proceedings in Theoretical Computer Science*, pages 27–41, 2017.

18   Salvador Lucas. A semantic approach to the analysis of rewriting-based systems. In Fabio Fioravanti and John P. Gallagher, editors, *Pre-Proceedings of the 27th International Symposium on Logic-Based Program Synthesis and Transformation*, 2017. *CoRR*, abs/1709.05095.

19   Salvador Lucas and Raúl Gutiérrez. A semantic criterion for proving infeasibility in conditional rewriting. In Beniamino Accattoli and Bertram Felgenhauer, editors, *Proceedings of the 6th International Workshop on Confluence*, pages 15–20, 2017.

20   Salvador Lucas and Raúl Gutiérrez. Automatic synthesis of logical models for order-sorted first-order theories. *Journal of Automated Reasoning*, 60(4):465–01, 2018.

21   Salvador Lucas, Claude Marché, and José Meseguer. Operational termination of conditional term rewriting systems. *Information Processing Letters*, 95(4):446–453, 2005.

22   Massimo Marchiori. Unravelings and ultra-properties. In Michael Hanus and Mario Rodríguez-Artalejo, editors, *Proceedings of the 5th International Conference on Algebraic and Logic Programming*, volume 1139 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 1996.

23   Aart Middeldorp and Erik Hamoen. Completeness results for basic narrowing. *Applicable Algebra in Engineering, Communication and Computing*, 5:213–253, 1994.

24   Masanori Nagashima, Masahiko Sakai, and Toshiki Sakabe. Determinization of conditional term rewriting systems. *Theoretical Computer Science*, 464:72–89, 2012.

25   Naoki Nishida, Takayuki Kuroda, Makishi Yanagisawa, and Karl Gmeiner. CO3: a COnverter for proving COnfluence of COnditional TRSs. In Ashish Tiwari and Takahito Aoto, editors, *Proceedings of the 4th International Workshop on Confluence*, page 42, 2015.

26   Naoki Nishida, Adrián Palacios, and Germán Vidal. Reversible computation in term rewriting. *Journal of Logical and Algebraic Methods in Programming*, 94:128–149, 2018.

27   Naoki Nishida and Germán Vidal. Program inversion for tail recursive functions. In Manfred Schmidt-Schauß, editor, *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications*, volume 10 of *Leibniz International Proceedings in Informatics*, pages 283–298. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2011.

28   Naoki Nishida and Germán Vidal. Computing more specific versions of conditional rewriting systems. In Elvira Albert, editor, *Revised Selected Papers of the 22nd International Symposium on Logic-Based Program Synthesis and Transformation*, volume 7844 of *Lecture Notes in Computer Science*, pages 137–154. Springer, 2013.

29   Naoki Nishida and Germán Vidal. A finite representation of the narrowing space. In Gopal Gupta and Ricardo Peña, editors, *Revised Selected Papers of the 23rd International Symposium on Logic-Based Program Synthesis and Transformation*, volume 8901 of *Lecture Notes in Computer Science*, pages 54–71. Springer, 2014.

30   Naoki Nishida and Germán Vidal. A framework for computing finite SLD trees. *Journal of Logic and Algebraic Methods in Programming*, 84(2):197–217, 2015.

31   Naoki Nishida, Makishi Yanagisawa, and Karl Gmeiner. On proving confluence of conditional term rewriting systems via the computationally equivalent transformation. In Takahito Aoto and Delia Kesner, editors, *Proceedings of the 3rd International Workshop on Confluence*, pages 24–28, 2014.

32   Enno Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, 2002.

33   Catuscia Palamidessi. Algebraic properties of idempotent substitutions. In Mike Paterson, editor, *Proceedings of the 17th International Colloquium on Automata, Languages and*

Programming, volume 443 of *Lecture Notes in Computer Science*, pages 386–399. Springer, 1990.

34 James R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity and associativity. *Journal of the ACM*, 21(4):622–642, 1974.

35 Thomas Sternagel and Aart Middeldorp. Conditional confluence (system description). In Gilles Dowek, editor, *Proceedings of the Joint International Conference on Rewriting and Typed Lambda Calculi*, volume 8560 of *Lecture Notes in Computer Science*, pages 456–465. Springer, 2014.

36 Thomas Sternagel and Aart Middeldorp. Infeasible conditional critical pairs. In Ashish Tiwari and Takahito Aoto, editors, *Proceedings of the 4th International Workshop on Confluence*, pages 13–17, 2015.

37 Taro Suzuki, Aart Middeldorp, and Tetsuo Ida. Level-confluence of conditional rewrite systems with extra variables in right-hand sides. In Jieh Hsiang, editor, *Proceedings of the 6th International Conference on Rewriting Techniques and Applications*, volume 914 of *Lecture Notes in Computer Science*, pages 179–193. Springer, 1995.

A   Example of Innermost Narrowing and Constructor-based Rewriting

▶ **Example 26.** Consider $\mathcal{R}_1$ in Section 1 again. We have infinitely many leftmost innermost narrowing derivations starting from $\mathsf{e}(x) \twoheadrightarrow \mathsf{true}$:

- $(\mathsf{e}(x) \twoheadrightarrow \mathsf{true}) \overset{li}{\rightsquigarrow}_{\{x \mapsto 0\}, \mathcal{R}_1} (\mathsf{true} \twoheadrightarrow \mathsf{true}) \overset{li}{\rightsquigarrow}_{id, \mathcal{R}_1} \top$,

- $(\mathsf{e}(x) \twoheadrightarrow \mathsf{true}) \overset{li}{\rightsquigarrow}_{\{x \mapsto \mathsf{s}(x_1)\}, \mathcal{R}_1} (\mathsf{o}(x_1) \twoheadrightarrow \mathsf{true}) \ \& \ (\mathsf{true} \twoheadrightarrow \mathsf{true}) \overset{li}{\rightsquigarrow}_{\{x_1 \mapsto 0\}, \mathcal{R}_1} (\mathsf{false} \twoheadrightarrow \mathsf{true}) \ \& \ (\mathsf{true} \twoheadrightarrow \mathsf{true})$,

- $(\mathsf{e}(x) \twoheadrightarrow \mathsf{true}) \overset{li}{\rightsquigarrow}_{\{x \mapsto \mathsf{s}(x_1)\}, \mathcal{R}_1} (\mathsf{o}(x_1) \twoheadrightarrow \mathsf{true}) \ \& \ (\mathsf{true} \twoheadrightarrow \mathsf{true}) \overset{li}{\rightsquigarrow}_{\{x_1 \mapsto \mathsf{s}(x_2)\}, \mathcal{R}_1} (\mathsf{e}(x_2) \twoheadrightarrow \mathsf{true}) \ \& \ (\mathsf{true} \twoheadrightarrow \mathsf{true}) \ \& \ (\mathsf{true} \twoheadrightarrow \mathsf{true}) \overset{li}{\rightsquigarrow}_{\{x_2 \mapsto 0\}, \mathcal{R}_1} (\mathsf{true} \twoheadrightarrow \mathsf{true}) \ \& \ (\mathsf{true} \twoheadrightarrow \mathsf{true}) \ \& \ (\mathsf{true} \twoheadrightarrow \mathsf{true}) \overset{li}{\rightsquigarrow}_{id, \mathcal{R}_1} \top \ \& \ (\mathsf{true} \twoheadrightarrow \mathsf{true}) \ \& \ (\mathsf{true} \twoheadrightarrow \mathsf{true}) \overset{li}{\rightsquigarrow}_{id, \mathcal{R}_1} \top \ \& \ \top \ \& \ (\mathsf{true} \twoheadrightarrow \mathsf{true}) \overset{li}{\rightsquigarrow}_{id, \mathcal{R}_1} \top \ \& \ \top \ \& \ \top$,

- ...

The following leftmost constructor-based rewriting derivations correspond to the above narrowing derivations, respectively:

- $(\mathsf{e}(0) \twoheadrightarrow \mathsf{true}) \overset{lc}{\rightarrow}_{\mathcal{R}_1} (\mathsf{true} \twoheadrightarrow \mathsf{true}) \overset{lc}{\rightarrow}_{\mathcal{R}_1} \top$,

- $(\mathsf{e}(\mathsf{s}(0)) \twoheadrightarrow \mathsf{true}) \overset{lc}{\rightarrow}_{\mathcal{R}_1} (\mathsf{o}(0) \twoheadrightarrow \mathsf{true}) \ \& \ (\mathsf{true} \twoheadrightarrow \mathsf{true}) \overset{lc}{\rightarrow}_{\mathcal{R}_1} (\mathsf{false} \twoheadrightarrow \mathsf{true}) \ \& \ (\mathsf{true} \twoheadrightarrow \mathsf{true})$,

- $(\mathsf{e}(\mathsf{s}(\mathsf{s}((0)))) \twoheadrightarrow \mathsf{true}) \overset{lc}{\rightarrow}_{\mathcal{R}_1} (\mathsf{o}(\mathsf{s}(0)) \twoheadrightarrow \mathsf{true}) \ \& \ (\mathsf{true} \twoheadrightarrow \mathsf{true}) \overset{lc}{\rightarrow}_{\mathcal{R}_1} (\mathsf{e}(0) \twoheadrightarrow \mathsf{true}) \ \& \ (\mathsf{true} \twoheadrightarrow \mathsf{true}) \ \& \ (\mathsf{true} \twoheadrightarrow \mathsf{true}) \overset{lc}{\rightarrow}_{\mathcal{R}_1} (\mathsf{true} \twoheadrightarrow \mathsf{true}) \ \& \ (\mathsf{true} \twoheadrightarrow \mathsf{true}) \ \& \ (\mathsf{true} \twoheadrightarrow \mathsf{true}) \overset{lc}{\rightarrow}_{\mathcal{R}_1} \top \ \& \ (\mathsf{true} \twoheadrightarrow \mathsf{true}) \ \& \ (\mathsf{true} \twoheadrightarrow \mathsf{true}) \overset{lc}{\rightarrow}_{\mathcal{R}_1} \top \ \& \ \top \ \& \ (\mathsf{true} \twoheadrightarrow \mathsf{true}) \overset{lc}{\rightarrow}_{\mathcal{R}_1} \top \ \& \ \top \ \& \ \top$,

- ...

B   Proofs of Theorems

In this appendix, we show proofs of Theorems 4, 6, 7, 19, and 22.

▶ **Theorem 4.** *Let $\mathcal{R}$ be a constructor SDCTRS, T a goal clause, and $U \in \mathcal{T}(\{\top, \&\})$.*

1. *If $T \overset{li}{\rightsquigarrow}{}^*_{\sigma, \mathcal{R}} U$, then $\sigma T \overset{lc}{\rightarrow}{}^*_{\mathcal{R}} U$ (i.e., $\sigma s \overset{lc}{\rightarrow}{}^*_{\mathcal{R}} \sigma t$ for all goals $s \twoheadrightarrow t$ in T).*

2. *For a constructor substitution θ, if $\theta T \overset{lc}{\rightarrow}{}^*_{\mathcal{R}} U$, then there exists an idempotent constructor substitution σ such that $T \overset{li}{\rightsquigarrow}{}^*_{\sigma, \mathcal{R}} U$ and $\sigma \le \theta$.*

Proof. The first claim can be straightforwardly proved by induction on the length of $T \overset{li}{\underset{\sigma, \mathcal{R}}{\leadsto}}^* U$. In [28], the second claim is proved for a constructor SDCTRS $\mathcal{R}$ such that for each rule $\ell \to r \Leftarrow s_1 \twoheadrightarrow t_1 \mathbin{\&} \cdots \mathbin{\&} s_k \twoheadrightarrow t_k$, all $t_1, \ldots, t_k$ are constructor terms. Any rule $\ell \to r \Leftarrow s_1 \twoheadrightarrow t_1 \mathbin{\&} \cdots \mathbin{\&} s_k \twoheadrightarrow t_k$ is not used in $\overset{c}{\to}_{\mathcal{R}}$ if there exists some i such that t_i contains a defined symbol. In addition, in the proof, $\mathcal{R}$ does not have to be deterministic or a 3-CTRS. For this reason, the proof in [28, Lemma 17] can be a proof of this theorem. ◀

We show some lemmas to prove Theorem 6.

▶ **Lemma 27.** *Let $\mathcal{R}$ be a constructor SDCTRS over a signature $\mathcal{F}$ such that $\mathcal{R} = \mathcal{R}_0 \uplus \{\ell \to r \Leftarrow C\}$ where r is not a constructor term of $\mathcal{R}$. Let $\mathcal{R}' = \mathcal{R}_0 \cup \{\ell \to x \Leftarrow C \mathbin{\&} r \twoheadrightarrow x\}$, where x is a fresh variable. Then, $\mathcal{R}'$ is a constructor SDCTRS over $\mathcal{F}$ and is equivalent to $\mathcal{R}$ w.r.t. $\overset{c}{\to}$ and $\overset{i}{\leadsto}$.*

Proof. By definition, it is clear that $\mathcal{D}_{\mathcal{R}} = \mathcal{D}_{\mathcal{R}'}$. The remaining properties for equivalence w.r.t. $\overset{c}{\to}$ and $\overset{i}{\leadsto}$ can straightforwardly proved by induction on the numbers of rewriting and narrowing steps, respectively. ◀

▶ **Lemma 28.** *Let $\mathcal{R}$ be a constructor SDCTRS over a signature $\mathcal{F}$ such that $\mathcal{R} = \mathcal{R}_0 \cup \{\ell \to r \Leftarrow C_1 \mathbin{\&} s_i \twoheadrightarrow t_i \mathbin{\&} C_2\}$ where r is a constructor term, $p \neq \varepsilon$, and t_i is a ground normal form of $\mathcal{R}_u$ but not a constructor term. Let $\mathcal{R}' = \mathcal{R}_0 \cup \{\ell \to r \Leftarrow C_1 \mathbin{\&} s_i \twoheadrightarrow x \mathbin{\&} t_i \twoheadrightarrow x \mathbin{\&} C_2\}$, where x is a fresh variable. Then, $\mathcal{R}'$ is a constructor SDCTRS over $\mathcal{F}$ and is equivalent to $\mathcal{R}$ w.r.t. $\overset{c}{\to}$ and $\overset{i}{\leadsto}$.*

Proof. By definition, it is clear that $\mathcal{D}_{\mathcal{R}} = \mathcal{D}_{\mathcal{R}'}$. Since t_i contains a defined symbol, t_i is a ground normal form of $\mathcal{R}_u$ and there is no rule $\ell' \to r' \Leftarrow C'$ in $\mathcal{R}$ such that ℓ' matches a subterm of t_i. To use $\ell \to r \Leftarrow C_1 \mathbin{\&} s_i \twoheadrightarrow t_i \mathbin{\&} C_2$ in a constructor-based rewriting to a term in $\mathcal{T}(\{\top, \mathbin{\&}\})$, an instance of $s_i \twoheadrightarrow t_i$ should be reduced to a term in $\mathcal{T}(\{\top, \mathbin{\&}\})$. However, such a reduction is impossible for constructor-based rewriting because $t_i \twoheadrightarrow t_i$ cannot be rewritten or narrowed to $\top$. For this reason, $\ell \to r \Leftarrow C_1 \mathbin{\&} s_i \twoheadrightarrow t_i \mathbin{\&} C_2$ is never used in constructor-based rewriting or innermost narrowing to a term in $\mathcal{T}(\{\top, \mathbin{\&}\})$. For the same reason, $\ell \to r \Leftarrow C_1 \mathbin{\&} s_i \twoheadrightarrow x \mathbin{\&} t_i \twoheadrightarrow x \mathbin{\&} C_2 \in \mathcal{R}'$ is never used in constructor-based rewriting of $\mathcal{R}'$ to terms in $\mathcal{T}(\{\top, \mathbin{\&}\})$, either. Therefore, it is clear that for a goal clause T and a term $U \in \mathcal{T}(\{\top, \mathbin{\&}\})$, (a) $T \overset{lc}{\to}_{\mathcal{R}}^* U$ if and only if $T \overset{lc}{\to}_{\mathcal{R}'}^* U$, and (b) $T \overset{li}{\underset{\theta, \mathcal{R}}{\leadsto}}^* U$ if and only if $T \overset{li}{\underset{\theta, \mathcal{R}'}{\leadsto}}^* U$. ◀

▶ **Lemma 29.** *Let $\mathcal{R}$ be a constructor SDCTRS over a signature $\mathcal{F}$ such that $\mathcal{R} = \mathcal{R}_0 \cup \{\ell \to r \Leftarrow C_1 \mathbin{\&} s_i[s']_p \twoheadrightarrow t_i \mathbin{\&} C_2\}$ where r is a constructor term, $p \neq \varepsilon$, and s' is rooted by a defined symbol. Let $\mathcal{R}' = \mathcal{R}_0 \cup \{\ell \to r \Leftarrow C_1 \mathbin{\&} s' \twoheadrightarrow x \mathbin{\&} s_i[x]_p \twoheadrightarrow t_i \mathbin{\&} C_2\}$, where x is a fresh variable. Then, $\mathcal{R}'$ is a constructor SDCTRS over $\mathcal{F}$ and is equivalent to $\mathcal{R}$ w.r.t. $\overset{c}{\to}$ and $\overset{i}{\leadsto}$.*

Proof. By definition, it is clear that $\mathcal{D}_{\mathcal{R}} = \mathcal{D}_{\mathcal{R}'}$. The remaining properties for equivalence w.r.t. $\overset{c}{\to}$ and $\overset{i}{\leadsto}$ can straightforwardly proved by induction on the numbers of rewriting and narrowing steps, respectively. ◀

▶ **Theorem 6.** *Let $\mathcal{R}$ be a constructor SDCTRS over a signature $\mathcal{F}$. Then, $Pc(\mathcal{R})$ is a pc-CTRS over $\mathcal{F}$ and is equivalent to $\mathcal{R}$ w.r.t. $\overset{c}{\to}$ and $\overset{i}{\leadsto}$.*

Proof. By definition, it is clear that $Pc(\mathcal{R})$ is a pc-CTRS over the signature $\mathcal{F}$ and $\mathcal{D}_{\mathcal{R}} = \mathcal{D}_{Pc(\mathcal{R})}$. The remaining properties can be proved by Lemmas 27, 28, and 29, and [26, Theorem 26]. ◀

▶ **Theorem 7.** *Let $\mathcal{R}$ be a pc-CTRS over a signature $\mathcal{F}$. Then, $Trs(\mathcal{R})$ is a constructor TRS over $\mathcal{F} \cup \{\twoheadrightarrow, \top, \&\}$ and is equivalent to $\mathcal{R}$ w.r.t. $\overset{c}{\to}$ and $\overset{i}{\rightsquigarrow}$.*

Proof. By definition, it is clear that (1) $Trs(\mathcal{R})$ is a constructor TRS over $\mathcal{F} \cup \{\twoheadrightarrow, \top, \&\}$, (2) $\mathcal{D}_{\mathcal{R}} = \mathcal{D}_{Trs(\mathcal{R})}$, (3) $(s \twoheadrightarrow t) \overset{c}{\to}_{\mathcal{R}} T$ if and only if $(s \twoheadrightarrow t) \overset{c}{\to}_{Trs(\mathcal{R})} T$ and (4) $(s \twoheadrightarrow t) \overset{i}{\rightsquigarrow}_{\theta, \mathcal{R}} T$ if and only if $(s \twoheadrightarrow t) \overset{i}{\rightsquigarrow}_{\theta, Trs(\mathcal{R})} T$. Using (3), we can prove that for a goal clause T and a term $U \in \mathcal{T}(\{\top, \&\})$, $T \overset{lc}{\to}{}^*_{\mathcal{R}} U$ if and only if $T \overset{lc}{\to}{}^*_{\mathcal{R}'} U$, by induction on the number of rewriting steps. Using (4), we can prove that for a goal clause T and a term $U \in \mathcal{T}(\{\top, \&\})$, $T \overset{li}{\rightsquigarrow}{}^*_{\theta, \mathcal{R}} U$ if and only if $T \overset{li}{\rightsquigarrow}{}^*_{\theta, \mathcal{R}'} U$, by induction on the number of narrowing steps.   ◀

▶ **Theorem 19.** *Let T be a goal clause without $\top$. Then, $[\![L(SSG(\mathcal{R}, T), \Gamma_T)]\!] = Suc(\overset{i}{\rightsquigarrow}_{\mathcal{R}}, T)$ up to variable renaming.*

Proof. Thanks to Theorems 12, 13, and 14, a constructor substitution θ for $T \overset{i}{\rightsquigarrow}{}^*_{\theta, \mathcal{R}} U \in \mathcal{T}(\{\top, \&\})$ can be obtained by (1) splitting, (2) flattening, and (3) narrowing applied to goals of the form $f(x_1, \ldots, x_n) \twoheadrightarrow u$ such that $u \in (\mathcal{V} \setminus \{x_1, \ldots, x_n\}) \cup \mathcal{T}(\mathcal{C}_{\mathcal{R}})$ and $x_1, \ldots, x_n$ are distinct variables. The application of these operations is exactly the same as the application of production rules in $SSG(\mathcal{R}, T)$. Therefore, this theorem holds.   ◀

The following lemma is used to prove Theorem 22.

▶ **Lemma 30.** *Let $\mathcal{G}$ be an SSG $(\Gamma_{T_0}, \mathcal{N}, \mathcal{P})$, $\Gamma_T \in \mathcal{N}$, $x \in Var(T)$, and $RTG(\mathcal{G}, T, x)$ be constructed. Then, $\{\xi \theta x \mid \theta \in [\![L(\mathcal{G}, \Gamma_T)]\!], \xi \in Subst(\mathcal{C}_{\mathcal{R}}), \mathcal{D}om(\eta) = \mathcal{V}ar(\theta x)\} \subseteq L(RTG(\mathcal{G}, T, x))$.*

Proof. Suppose that e is generated by n steps of applying production rules obtained from $\mathcal{P}|_{\Gamma_T}$. Then, it is easy to prove this lemma by induction on n.   ◀

The converse inclusion in Lemma 30 does not hold in general even if $[\![L(\mathcal{G}, \Gamma_T)]\!]$ is a set of ground substitutions.

▶ **Example 31.** Consider an SSG $\mathcal{G}_4 = (\Gamma_{x \twoheadrightarrow a}, \{\Gamma_{x \twoheadrightarrow a}, \Gamma_{z \twoheadrightarrow b}\}, \{ \Gamma_{x \twoheadrightarrow a} \to \text{REC}(\Gamma_{z \twoheadrightarrow b}, \{y \mapsto z\}) \bullet \{x \mapsto c(y, y)\}, \Gamma_{z \twoheadrightarrow b} \to \{z \mapsto a\} \mid \{z \mapsto b\} \})$. For goal $x \twoheadrightarrow a$ and variable x, we have the regular tree grammar $RTG(\mathcal{G}_4, x \twoheadrightarrow a, x) = (\Gamma^x_{x \twoheadrightarrow a}, \{\Gamma^x_{x \twoheadrightarrow a}, \Gamma^z_{z \twoheadrightarrow b}\}, \{ \Gamma^x_{x \twoheadrightarrow a} \to c(\Gamma^z_{z \twoheadrightarrow b}, \Gamma^z_{z \twoheadrightarrow b}), \Gamma^z_{z \twoheadrightarrow b} \to a \mid b \})$. The term $c(a, b)$ is included in $L(RTG(\mathcal{G}_4, x \twoheadrightarrow a, x), \Gamma^x_{x \twoheadrightarrow a})$, but there is no substitution θ such that $\theta x = c(a, b)$ and $\sigma \le \theta$ for some σ in $[\![L(\mathcal{G}_4, \Gamma_{x \twoheadrightarrow a})]\!] = \{ \{x \mapsto c(a, a)\}, \{x \mapsto c(b, b)\} \}$.

▶ **Theorem 22.** *Let $\mathcal{G}$ be an SSG $(\Gamma_{T_0}, \mathcal{N}, \mathcal{P})$, $\Gamma_{T_1}, \Gamma_{T_2} \in \mathcal{N}$, $x \in \mathcal{V}$, $x_1 \in \mathcal{V}ar(T_1)$, $x_2 \in \mathcal{V}ar(T_2)$, $RTG(\mathcal{G}, T_1, x_1), RTG(\mathcal{G}, T_2, x_2)$ be constructed, and δ_1, δ_2 be renamings such that $\mathcal{VR}an(\delta_i) = \mathcal{V}ar(T_i)$ and $\delta_i x = x_i$ for $i = 1, 2$. If $L(RTG(\mathcal{G}, T_1, x_1)) \cap L(RTG(\mathcal{G}, T_2, x_2)) = \emptyset$, then $\{\!| \text{REC}(\Gamma_{T_1}, \delta_1) \& \text{REC}(\Gamma_{T_2}, \delta_2) |\!\}_{\mathcal{G}} = \{\!| \varnothing |\!\}_{\mathcal{G}}$.*

Proof. We proceed by contradiction. Assume that $L(RTG(\mathcal{G}, T_1, x_1)) \cap L(RTG(\mathcal{G}, T_2, x_2)) = \emptyset$ and $\{\!| \text{REC}(\Gamma_{T_1}, \delta_1) \& \text{REC}(\Gamma_{T_2}, \delta_2) |\!\}_{\mathcal{G}} \ne \{\!| \varnothing |\!\}_{\mathcal{G}}$. Then, there exists a constructor substitution $\theta \in \{\!| \text{REC}(\Gamma_{T_1}, \delta_1) \& \text{REC}(\Gamma_{T_2}, \delta_2) |\!\}_{\mathcal{G}}$, and hence there exist constructor substitutions θ_1, θ_2 such that $\theta_1 \in \{\!| \Gamma_{T_1} |\!\}_{\mathcal{G}}$, $\theta_2 \in \{\!| \Gamma_{T_2} |\!\}_{\mathcal{G}}$, and $\theta = (\theta_1 \cdot \delta_1) \Uparrow (\theta_2 \cdot \delta_2)$. Thus, it follows from Lemma 30 that $\xi \theta x \in L(RTG(\mathcal{G}, T_1, x_1)) \cap L(RTG(\mathcal{G}, T_2, x_2))$ for some $\xi \in Subst(\mathcal{C}_{\mathcal{R}})$. This contradicts the assumption.   ◀

Homogeneity Without Loss of Generality

Paweł Parys

University of Warsaw
Warsaw, Poland
parys@mimuw.edu.pl

---- **Abstract** --

We consider higher-order recursion schemes as generators of infinite trees. A sort (simple type) is called homogeneous when all arguments of higher order are taken before any arguments of lower order. We prove that every scheme can be converted into an equivalent one (i.e, generating the same tree) that is homogeneous, that is, uses only homogeneous sorts. Then, we prove the same for safe schemes: every safe scheme can be converted into an equivalent safe homogeneous scheme. Furthermore, we compare two definition of safe schemes: the original definition of Damm, and the modern one. Finally, we prove a lemma which illustrates usefulness of the homogeneity assumption. The results are known, but we prove them in a novel way: by directly manipulating considered schemes.

1   Introduction

Higher-order recursion schemes (*schemes* in short) are used to faithfully represent the control flow of programs in languages with higher-order functions. This formalism is equivalent via direct translations to simply-typed λY-calculus [18] and to higher-order OI grammars [9, 15]. Collapsible pushdown systems [10] and ordered tree-pushdown systems [7] are other equivalent formalisms. Schemes cover some other models such as indexed grammars [1] and ordered multi-pushdown automata [4]. We consider schemes as generators of infinite trees, so we say that two schemes are equivalent if they generate the same tree. Likewise, we say that two classes of schemes are equi-expressive, if for every scheme in one of the classes there exists an equivalent scheme in the other class.

A sort (simple type) is called homogeneous when all arguments of higher order are taken before any arguments of lower order; a scheme is homogeneous when it uses only homogeneous sorts. Homogeneous schemes should not be confused with safe schemes. The safety assumption was first introduced implicitly by Damm [9]. His restriction was that when an argument of some order is applied to a function, then all arguments of greater or the same order have to be applied as well. A modern definition of safety (introduced by Knapik, Niwiński, Urzyczyn [14]) is slightly different: it says that a subterm of some order cannot use parameters of a strictly smaller order. We remark that some authors, while defining

safe schemes, require that they are also homogeneous [9, 13, 14], while other authors do not impose this requirement [3, 6]. In this paper we treat homogeneity separately from safety.

The goal of this paper is to compare the aforementioned notions, and to give simple translations between equi-expressive classes of schemes. The main equi-expressivity result says that every scheme can be converted into a homogeneous scheme that is equivalent, and remains of the same order. This was shown by Broadbent in his PhD thesis [5, Section 3.4], and was never published. Furthermore, it is easy to see that the Damm's definition of safety is more restrictive than the modern one. On the other hand, it was observed by Carayol and Serre [6] that every scheme that is safe according to the modern definition can be turned into an equivalent scheme that is safe according to Damm's definition. Likewise, it was shown by Blum [2] (his paper dates back to 2009, when it was shared on his personal website, but was published on arXiv only in 2017), and independently by Carayol and Serre [6], that every safe scheme (without the homogeneity assumption) can be converted into an equivalent safe scheme that is homogeneous, and remains of the same order.

All the proofs for safe schemes follow the same idea: they inspect the equivalence between safe schemes and higher-order pushdown automata. It is observed that while translating from safe schemes to higher-order pushdown automata, schemes can comply with a less restrictive definition; simultaneously, when translating from automata to schemes, it is easy to fulfill additional requirements on the scheme.

The proof of Broadbent, dealing with schemes that need not to be safe, is even more complicated. Arbitrary schemes are equivalent to collapsible pushdown automata, a generalization of higher-order pushdown automata. We can see, though, that the only known translation from collapsible pushdown automata to recursion schemes [10] results in schemes that are not homogeneous. The actual proof consists of three steps. First, it is observed that already while translating a scheme to a collapsible pushdown automaton, the resulting automaton is of a special shape. Then, such an automaton is further modified (without changing the generated tree), so that it gains some additional properties. Finally, it is observed that for the particular automata obtained this way, the translation from automata to schemes can be altered so that the resulting schemes are homogeneous.

We reprove the above results: we give a simple transformation changing any scheme to an equivalent homogeneous scheme, and another simple transformation changing any safe scheme to a scheme that is safe according to the more restrictive definition of Damm, and moreover homogeneous.

Both our proofs (the one for general schemes, and the one for safe schemes) do not use any detour through automata; we directly show how to syntactically modify a scheme so that it becomes homogeneous. Roughly, in the case of general schemes we artificially increase orders of some arguments, while in the case of safe schemes we split complex rules into multiple simpler rules, and we reorder arguments. Our direct approach has the advantage that it is more transparent and it sheds some light on the nature of the homogeneity assumption (conversely to the previous proofs: while translating a scheme to an automaton and then back to a scheme, we obtain a scheme of a completely different shape than the original one).

In order to give a full picture we have to recall here the result that there is a scheme that is not equivalent to any safe scheme [16]. We thus have two groups of equi-expressive classes: "unsafe" schemes, either homogeneous or not, and safe schemes, either according to the Damm's definition or to the modern definition, and either homogeneous or not.

In addition to the above results, in the final section we prove a simple lemma, which illustrates usefulness of the homogeneity assumption.

2 Preliminaries

Infinitary λ-calculus. The set of *sorts* (aka. simple types) is defined by induction: o is a sort, and if α and β are sorts, then $\alpha \to \beta$ is a sort. We omit brackets on the right of an arrow, so, for example, $o \to (o \to o)$ is abbreviated to $o \to o \to o$. Notice that every sort can be written in the form $\alpha_1 \to \cdots \to \alpha_s \to o$.

The *order* of a sort γ, denoted $ord(\gamma)$, is defined by induction on the structure of γ: $ord(o) = 0$, and $ord(\alpha \to \beta) = \max(ord(\alpha) + 1, ord(\beta))$. We observe that $ord(\alpha_1 \to \cdots \to \alpha_s \to o) = 1 + \max(ord(\alpha_1), \ldots, ord(\alpha_s))$ whenever $s \geq 1$.

A sort $\alpha_1 \to \cdots \to \alpha_s \to o$ is *homogeneous* if $ord(\alpha_1) \geq \cdots \geq ord(\alpha_s)$ and all $\alpha_1, \ldots, \alpha_s$ are homogeneous. An equivalent definition says that the sort o is homogeneous, and a sort $\alpha \to \beta$ is homogeneous if $ord(\alpha) = ord(\alpha \to \beta) - 1$ and α, β are homogeneous.

While defining λ-terms, we assume existence of the following sets:
- Σ – a set of symbols (alphabet), and
- $\mathcal{V}$ – a set of variables with assigned sorts; we write $x^\alpha, y^\alpha, z^\alpha, \ldots$ for variables of sort α.

We consider infinitary, sorted λ-calculus. *Infinitary λ-terms* (or just λ-*terms*) are defined by coinduction (for an introduction to coinductive definitions and proofs see, e.g., Czajka [8]), according to the following rules:
- node constructor – if $K_1^o, \ldots, K_r^o$ are λ-terms, then $(a\langle K_1^o, \ldots, K_r^o \rangle)^o$ is a λ-term, for every $a \in \Sigma$,
- variable – every variable $x^\alpha \in \mathcal{V}$ is a λ-term,
- application – if $K^{\alpha \to \beta}$ and L^α are λ-terms, then $(K^{\alpha \to \beta} L^\alpha)^\beta$ is a λ-term, and
- λ-binder – if K^β is a λ-term and $x^\alpha \in \mathcal{V}$ is a variable, then $(\lambda x^\alpha.K^\beta)^{\alpha \to \beta}$ is a λ-term.

We naturally identify λ-terms differing only in names of bound variables. We often omit sort annotations of λ-terms, but we keep in mind that every λ-term (and every variable) has a particular sort. The set of *free variables* of a λ-term M, denoted $FV(M)$, is defined as usual. A λ-term M is *closed* if $FV(M) = \emptyset$. We assume that in $\mathcal{V}$ there are always some fresh variables of every sort, not appearing in λ-terms under consideration.

The *order* of a λ-term M, written $ord(M)$, is just the order of its sort. The *complexity* of a λ-term M is the smallest number $m \in \mathbb{N} \cup \{\infty\}$ such that all subterms of M are of order at most m.

Reductions. By $M[N/x]$ (where we require that N is of the same sort as x) we denote the λ-term obtained by substituting N for x. This is by definition a capture-avoiding substitution, which means that free variables of N are not captured by λ-binders in M; this is achieved by appropriately renaming bound variables in M.

A *compatible closure* $\rightsquigarrow$ of a relation $\twoheadrightarrow$ is defined by induction according to the following rules:
- if $M \twoheadrightarrow N$, then $M \rightsquigarrow N$,
- if $K_j \rightsquigarrow K_j'$ for some $j \in \{1, \ldots, r\}$ and $K_i = K_i'$ for all $i \in \{1, \ldots, r\} \setminus \{j\}$, then $a\langle K_1, \ldots, K_r \rangle \rightsquigarrow a\langle K_1', \ldots, K_r' \rangle$,
- if $K \rightsquigarrow K'$, then $K L \rightsquigarrow K' L$,
- if $L \rightsquigarrow L'$, then $K L \rightsquigarrow K L'$, and
- if $K \rightsquigarrow K'$, then $\lambda x.K \rightsquigarrow \lambda x.K'$.

The relation $\to_\beta$ of *β-reduction* is defined as the compatible closure of the relation $\{((\lambda x.K) L, K[L/x])\}$. The relation $\to_\eta$ of *η-conversion* is defined as the compatible closure of the relation $\{(\lambda x.K x, K) \mid x \notin FV(K)\}$. We let $(\to_{\beta\eta}) = (\to_\beta) \cup (\to_\eta)$. As a restriction

of β-reduction, we define the relation $\xrightarrow{h}_\beta$ of *head β-reduction*: it contains all pairs of the form

$$((\lambda x.K)\, L\, P_1\, \ldots\, P_n, K[L/x]\, P_1\, \ldots\, P_n).$$

For relations $\rightsquigarrow$ and $\twoheadrightarrow$, by $(\rightsquigarrow) \circ (\twoheadrightarrow)$ we denote their composition, by $\rightsquigarrow^k$ (where $k \in \mathbb{N}$) the composition of $\rightsquigarrow$ with itself k times, and by $\rightsquigarrow^*$ the reflexive transitive closure of $\rightsquigarrow$. Moreover, $\rightsquigarrow^\infty$ is the infinitary closure of $\rightsquigarrow$, defined by coinduction, according to the following rules:

- if $M \rightsquigarrow^* a\langle K_1, \ldots, K_r\rangle$ and $K_i \rightsquigarrow^\infty K_i'$ for all $i \in \{1, \ldots, r\}$, then $M \rightsquigarrow^\infty a\langle K_1', \ldots, K_r'\rangle$,
- if $M \rightsquigarrow^* x$ then $M \rightsquigarrow^\infty x$,
- if $M \rightsquigarrow^* K\, L$, and $K \rightsquigarrow^\infty K'$, and $L \rightsquigarrow^\infty L'$, then $M \rightsquigarrow^\infty K'\, L'$, and
- if $M \rightsquigarrow^* \lambda x.K$ and $K \rightsquigarrow^\infty K'$, then $M \rightsquigarrow^\infty \lambda x.K'$.

Trees; Böhm Trees.   A *tree* is defined as a λ-term that is built using only node constructors, that is, not using variables, applications, nor λ-binders.

We consider Böhm trees only for closed λ-terms of sort o. For such a λ-term M, its *Böhm tree $BT(M)$* is defined by coinduction, as follows:

- if $M \xrightarrow{h}_\beta^* a\langle K_1, \ldots, K_r\rangle$ for some $a \in \Sigma$ and some λ-terms $K_1, \ldots, K_r$, then $BT(M) = a\langle BT(K_1), \ldots, BT(K_r)\rangle$;
- otherwise $BT(M) = \bot\langle\rangle$ (where $\bot \in \Sigma$ is a distinguished symbol).

With such a definition it is easy to see that for every M there is exactly one Böhm tree. It is a consequence of Kennaway, Klop, Sleep, de Vries [11] and Kennaway, van Oostrom, de Vries [12] that the Böhm tree does not change during $\beta\eta$-reductions.

▶ **Fact 1.** *If M and N are closed λ-terms of sort o and $M \rightarrow^\infty_{\beta\eta} N$, then $BT(M) = BT(N)$.*

Higher-Order Recursion Schemes.   A *higher-order recursion scheme* (or just a *scheme*) is a triple $\mathcal{G} = (\mathcal{N}, \mathcal{R}, X_0^o)$, where $\mathcal{N} \subseteq \mathcal{V}$ is a finite set of nonterminals, $X_0^o \in \mathcal{N}$ is a starting nonterminal, being of sort o, and $\mathcal{R}$ is a function that maps every nonterminal $X \in \mathcal{N}$ to a finite λ-term of the form $\lambda x_1. \cdots . \lambda x_s.M$, where

- the sorts of X and $\lambda x_1. \cdots . \lambda x_s.M$ are the same,
- $FV(M) \subseteq \mathcal{N} \cup \{x_1, \ldots, x_s\}$,
- M is of sort o, and
- M is a finite *applicative term*, that is, it does not contain any λ-binders.

We assume that elements of $\mathcal{N}$ are not used as bound variables, and that $\mathcal{R}(X)$ is not a nonterminal.[1] When $\mathcal{R}(X) = \lambda x_1. \cdots . \lambda x_s.M$, we say that $X\, x_1\, \ldots\, x_s \rightarrow M$ is a *rule* of $\mathcal{G}$, and M is its *right side*. The *order* of the scheme is defined as the maximum of orders of nonterminals in $\mathcal{N}$.

The infinitary λ-term *generated by* a scheme $\mathcal{G} = (\mathcal{N}, \mathcal{R}, X_0)$ from a λ-term M, denoted $\Lambda_\mathcal{G}(M)$, is defined as the limit of the following process starting from M: take any nonterminal X appearing in the current term, and replace it by $\mathcal{R}(X)$. We define $\Lambda(\mathcal{G}) = \Lambda_\mathcal{G}(X_0)$; observe that this is a closed λ-term of sort o and of complexity not greater than the order of the scheme. The *tree generated by* $\mathcal{G}$ is defined as $BT(\Lambda(\mathcal{G}))$.

---

[1] Without the last condition, it would be necessary to give a more complicated definition of $\Lambda(\mathcal{G})$. On the other hand, it is easy to ensure this condition, without changing the tree generated by the scheme.

We say that a scheme $\mathcal{G} = (\mathcal{N}, \mathcal{R}, X_0)$ is *homogeneous* if sorts of all nonterminals in $\mathcal{N}$ are homogeneous. Notice that then also the sort of every subterm of $\mathcal{R}(X)$ is homogeneous, for every nonterminal $X \in \mathcal{N}$.

▶ **Example 1.** Consider a scheme $\mathcal{G}_1$ with nonterminals Y_0^o, $Y_1^{o \to ((o \to o) \to o) \to o}$, $Y_2^{o \to o}$, and $Y_3^{o \to (o \to o) \to o}$, where Y_0 is starting; and rules

$$Y_0 \to Y_1 \left(b\langle c\langle\rangle\rangle \right) \left(Y_3 \left(c\langle\rangle \right) \right), \qquad\qquad Y_2\, x^o \to x\,,$$
$$Y_1\, x^o\, z^{(o \to o) \to o} \to a\langle z\, Y_2, Y_1 \left(b\langle x\rangle \right) \left(Y_3\, x \right)\rangle, \qquad\qquad Y_3\, x^o\, y^{o \to o} \to y\, x\,.$$

Then

$$\Lambda(\mathcal{G}_1) = M_1 \left(b\langle c\langle\rangle\rangle \right) \left((\lambda x^o.\lambda y^{o \to o}.y\, x) \left(c\langle\rangle \right) \right),$$

where M_1 is the unique λ-term such that

$$M_1 = \lambda x^o.\lambda z^{(o \to o) \to o}.a\langle z\, (\lambda x^o.x), M_1 \left(b\langle x\rangle \right) \left((\lambda x^o.\lambda y^{o \to o}.y\, x)\, x \right)\rangle\,.$$

We can see that $BT(\Lambda(\mathcal{G}_1)) = a\langle T_0, a\langle T_1, a\langle T_2, \ldots\rangle\rangle\rangle$, where $T_0 = c\langle\rangle$, and $T_{i+1} = b\langle T_i\rangle$ for $i \in \mathbb{N}$.

Notice that the sorts of Y_1 and of Y_3 are not homogeneous: the first parameter is of order 0, and the second of order 2 or 1.

3 Ensuring Homogeneity

We now prove our main theorem:

▶ **Theorem 2.** *For every scheme $\mathcal{G} = (\mathcal{N}, \mathcal{R}, X_0)$ one can construct in logarithmic space a homogeneous scheme $\mathcal{H}$ that is of the same order as $\mathcal{G}$ and such that $BT(\Lambda(\mathcal{H})) = BT(\Lambda(\mathcal{G}))$.*

Let us first present the general idea of the proof. Consider thus a nonterminal X with $\mathcal{R}(X) = \lambda x.\lambda y.K$, where $ord(x) < ord(y)$ (like Y_1 or Y_3 in Example 1). The sort of X is not homogeneous, as it does not satisfy $ord(x) \geq ord(y)$. How can we make it homogeneous?

One idea, which does not work, is to swap the order of x and y. The sort of $\lambda y.\lambda x.K$ is indeed homogeneous. Such a swap is problematic, though: possibly there are places where only one argument is given to X, corresponding to the parameter x (e.g., in Example 1 we always give only one argument to Y_3). When the parameters are swapped, we cannot pass a value of x to X, without passing a value of y.

There is another simple idea, which actually works. Namely, we should raise the order of x to $ord(y)$. How can we do that? Simply instead of passing to X an argument M of a low order $ord(x)$, we pass a function $\lambda z.M$ (of order $ord(y)$, higher than $ord(x)$), which ignores its argument z and returns M. On the other side, we change every use of x in K to $x\, N$, where N is an arbitrary λ-term of the same sort as z.

Notice that after such a modification of the sort of x, the order of $\lambda x.\lambda y.K$ remains as before the modification. This is very important: thanks to this property (orders of subterms do not change), we can perform the modification independently in every place. Moreover, as a side effect, also the order of the whole scheme remains unchanged.

There is one more difficulty to overcome, while proving the theorem. Namely, in λ-calculus it would be possible to simply write $\lambda z.M$ instead of M, whenever we wanted to convert M into a function returning M. This is not so trivial for schemes, as we cannot use λ-binders – we should use nonterminals instead. Say that we want to change the order of M from 0 (sort

o) to 1 (sort $o \to o$). To this end, we introduce a nonterminal S with $\mathcal{R}(S) = \lambda x^o.\lambda z^o.x$, and we write $S\,M$ instead of $\lambda z.M$ (that is, instead of M in the original scheme).

Notice, though, that the sort of the new nonterminal S has to be homogeneous as well. This means that using such a nonterminal S we can raise the order only by one, as we cannot have $\mathcal{R}(S) = \lambda x^o.\lambda z.x$ with $ord(z) > 0$. If we want to raise the order of M from 0 to 2 (or more), beside of S we need another nonterminal S' which raises the order from 1 to 2 (again only by one), etc. As we start now from sort $o \to o$, we should take $\mathcal{R}(S') = \lambda x_1^{o \to o}.\lambda z_1^{o \to o}.\lambda z^o.x_1\,z$. We then write $S'\,(S\,M)$ instead of M, which, after expanding S and S', equals

$$(\lambda x_1^{o \to o}.\lambda z_1^{o \to o}.\lambda z^o.x_1\,z)\,((\lambda x^o.\lambda z^o.x)\,M)\,.$$

This β-reduces (in three steps) to $\lambda z_1^{o \to o}.\lambda z^o.M$, thus it is a function of order 2 ignoring its arguments and returning M.

We now come to details. First, we define sorts γ_k; these will be sorts of the spare parameters (i.e., of z in the above explanation). The definition is by induction:

$$\gamma_0 = o, \qquad\qquad\qquad \gamma_k = \gamma_{k-1} \to o \qquad \text{for } k \geq 1.$$

For example, $\gamma_1 = o \to o$ and $\gamma_2 = (o \to o) \to o$. We see that $ord(\gamma_k) = k$ for all $k \in \mathbb{N}$.

Next, we have an operation $\mathbf{R}_k$, which says how to raise the order of a sort α to k. For every sort α and every $k \geq ord(\alpha)$ we define:

$$\mathbf{R}_k(\alpha) = \gamma_{k-1} \to \gamma_{k-2} \to \cdots \to \gamma_{ord(\alpha)} \to \alpha\,.$$

In particular $\mathbf{R}_{ord(\alpha)}(\alpha) = \alpha$. We see that $ord(\mathbf{R}_k(\alpha)) = k$. Basing on $\mathbf{R}_k$, we define, by induction, a transformation $\mathbf{H}$ changing an arbitrary sort into a homogeneous one:

$$\mathbf{H}(o) = o, \qquad\qquad \mathbf{H}(\alpha \to \beta) = \mathbf{R}_{ord(\alpha \to \beta)-1}(\mathbf{H}(\alpha)) \to \mathbf{H}(\beta).$$

Notice that $ord(\mathbf{H}(\alpha)) = ord(\alpha)$ for every sort α, and that $\mathbf{H}(\alpha)$ is indeed homogeneous.

Next, we come to transforming λ-terms. For every sort α appearing in the original scheme $\mathcal{G}$ (as a sort of a subterm of $\mathcal{R}(X)$ for some nonterminal $X \in \mathcal{N}$), and for every k such that $ord(\alpha) < k \leq ord(\mathcal{G})$, we add a nonterminal $S_{\alpha,k}$. Its sort is $\mathbf{R}_{k-1}(\mathbf{H}(\alpha)) \to \mathbf{R}_k(\mathbf{H}(\alpha))$. Recall that $\mathbf{R}_k(\mathbf{H}(\alpha)) = \gamma_{k-1} \to \mathbf{R}_{k-1}(\mathbf{H}(\alpha))$; let us also write $\mathbf{R}_{k-1}(\mathbf{H}(\alpha)) = \beta_1 \to \cdots \to \beta_s \to o$. Then the rule for $S_{\alpha,k}$ is

$$\mathcal{R}'(S_{\alpha,k}) = \lambda x.\lambda z.\lambda y_1.\cdots.\lambda y_s.x\,y_1\,\ldots\,y_s\,.$$

Here the sort of x is $\mathbf{R}_{k-1}(\mathbf{H}(\alpha))$, the sort of z is γ_{k-1}, and the sorts of $y_1,\ldots,y_s$ are $\beta_1,\ldots,\beta_s$, respectively.

Let again α be a sort appearing in $\mathcal{G}$, and let k be such that $ord(\alpha) \leq k \leq ord(\mathcal{G})$. The sort of a λ-term may be changed from $\mathbf{H}(\alpha)$ to $\mathbf{R}_k(\mathbf{H}(\alpha))$ by applying the following transformation, also called $\mathbf{R}_k$:

$$\mathbf{R}_k(M) = S_{\alpha,k}\,(S_{\alpha,k-1}\,\ldots(S_{\alpha,ord(\alpha)+1}\,M)\ldots)\,.$$

Here, by appending a nonterminal $S_{\alpha,i}$ we change the sort from $\mathbf{R}_{i-1}(\mathbf{H}(\alpha))$ to $\mathbf{R}_i(\mathbf{H}(\alpha))$; recall that $\mathbf{R}_{ord(\alpha)}(\mathbf{H}(\alpha)) = \mathbf{H}(\alpha)$.

We also need an opposite operation, which converts a function back to its value, by applying some arguments of sorts γ_k. First we define some nonterminals of such sorts: we fix a symbol $e \in \Sigma$, and for every $k < ord(\mathcal{G})$ we add a nonterminal U_k of sort γ_k, and we take:

$$\mathcal{R}'(U_0) = e\langle\rangle, \qquad\qquad \mathcal{R}'(U_k) = \lambda z^{\gamma_{k-1}}.e\langle\rangle \qquad \text{for } k \geq 1.$$

Clearly U_k has sort γ_k, for every $k \in \mathbb{N}$.

When N is of sort $\mathbf{R}_k(\mathbf{H}(\alpha))$, and $ord(\alpha) = n$ (the relation between k and n is $k \geq n$), we define

$$\mathbf{L}_n(N) = N \, U_{k-1} \, U_{k-2} \, \ldots \, U_n \, .$$

This λ-term is indeed of sort $\mathbf{H}(\alpha)$.

Using the above operations, we define a transformation changing the original scheme into a homogeneous one. Let us first describe this transformation informally. It works as follows. We first change the sort of every λ-term (i.e., every nonterminal, every variable, and every subterm of the right side of every rule) from α to $\mathbf{H}(\alpha)$. This causes a problem on applications, since to a function of sort $\mathbf{H}(\alpha \to \beta) = \mathbf{R}_{ord(\alpha \to \beta)-1}(\mathbf{H}(\alpha)) \to \mathbf{H}(\beta)$ we apply an argument of sort $\mathbf{H}(\alpha)$. We thus repair the argument by applying $\mathbf{R}_{ord(\alpha \to \beta)-1}(\cdot)$ to it. This also causes a problem on λ-binders and on variables: the new sort of a λ-binder $\lambda x^\alpha.K^\beta$ should be $\mathbf{H}(\alpha \to \beta) = \mathbf{R}_{ord(\alpha \to \beta)-1}(\mathbf{H}(\alpha)) \to \mathbf{H}(\beta)$, so the sort of the variable should be $\mathbf{R}_{ord(\alpha \to \beta)-1}(\mathbf{H}(\alpha))$; however, while using this variable, we expect that it will have sort $\mathbf{H}(\alpha)$. We thus apply $\mathbf{L}_{ord(\alpha)}(\cdot)$ to every place where the variable is used. There is no problem with nonterminals: every nonterminal simply changes its sort from α to $\mathbf{H}(\alpha)$.

We now define the transformation formally. A *raise environment* is a function Ω mapping some variable names to sorts, where we require that $\Omega(x^\alpha)$ equals $\mathbf{R}_k(\mathbf{H}(\alpha))$ for some $k \geq ord(\alpha)$. Intuitively, $\Omega(x^\alpha)$ is a new sort that the variable gets after the transformation. For a raise environment Ω (such that $FV(M) \subseteq dom(\Omega)$) we define $\mathbf{H}_\Omega(M)$ by coinduction on the structure of a λ-term M:

$$\mathbf{H}_\Omega(a\langle K_1, \ldots, K_r\rangle) = a\langle \mathbf{H}_\Omega(K_1), \ldots, \mathbf{H}_\Omega(K_r)\rangle \, .$$

$$\mathbf{H}_\Omega(x^\alpha) = \mathbf{L}_{ord(\alpha)}(x^{\Omega(x^\alpha)}) \qquad\qquad \text{if } x^\alpha \in \mathcal{V} \setminus \mathcal{N},$$

$$\mathbf{H}_\Omega(X^\alpha) = X^{\mathbf{H}(\alpha)} \qquad\qquad\qquad \text{if } X^\alpha \in \mathcal{N},$$

$$\mathbf{H}_\Omega(K \, L) = \mathbf{H}_\Omega(K) \, \mathbf{R}_{ord(K)-1}(\mathbf{H}_\Omega(L)) \, ,$$

$$\mathbf{H}_\Omega(\lambda x^\alpha.K) = \lambda x^{\alpha'}.\mathbf{H}_{\Omega[x^\alpha \mapsto \alpha']}(K) \, , \qquad \text{where } \alpha' = \mathbf{R}_{ord(\lambda x^\alpha.K)-1}(\mathbf{H}(\alpha)) \, .$$

Here by $\Omega[x^\alpha \mapsto \alpha']$ we mean the function that maps x^α to α', and every other variable $y \in dom(\Omega)$ to $\Omega(y)$. Notice that for M of sort α, the resulting λ-term $\mathbf{H}_\Omega(M)$ is of sort $\mathbf{H}(\alpha)$; in particular, in the case of an application with K of sort $\beta \to \gamma$, the sort of the function $\mathbf{H}_\Omega(K)$ being $\mathbf{H}(\beta \to \gamma) = \mathbf{R}_{ord(\beta \to \gamma)-1}(\mathbf{H}(\beta)) \to \mathbf{H}(\gamma)$ matches well with the sort of the argument, being $\mathbf{R}_{ord(\beta \to \gamma)-1}(\mathbf{H}(\beta))$.

The newly created scheme $\mathcal{H} = (\mathcal{N}', \mathcal{R}', X_0)$ is as follows. For every nonterminal $X^\alpha \in \mathcal{N}$, to $\mathcal{N}'$ we take a nonterminal $X^{\mathbf{H}(\alpha)}$, and we define $\mathcal{R}'(X^{\mathbf{H}(\alpha)}) = \mathbf{H}_\emptyset(\mathcal{R}(X^\alpha))$ (where $\emptyset$ is the raise environment with empty domain). Additionally in $\mathcal{N}'$ we have nonterminals $S_{\alpha,k}$ and U_k, with appropriate rules, as defined above.

▶ **Example 1** (continued). While applying our transformation to the scheme $\mathcal{G}_1$ from Example 1, we obtain a homogeneous scheme with the following rules (where we write S_i instead of $S_{o,i}$):

$$Y_0 \to Y_1 \, (S_2 \, (S_1 \, (b\langle c\langle\rangle\rangle))) \, (Y_3 \, (S_1 \, (c\langle\rangle))) \, ,$$

$$Y_1 \, x^{(o\to o)\to o\to o} \, z^{(o\to o)\to o} \to a\langle z \, Y_2, Y_1 \, (S_2 \, (S_1 \, (b\langle x \, U_1 \, U_0\rangle))) \, (Y_3 \, (S_1 \, (x \, U_1 \, U_0)))\rangle \, ,$$

$$Y_2 \, x^o \to x \, , \qquad\qquad\qquad S_1 \, x^o \, z^o \to x \, , \qquad\qquad\quad U_0 \to e\langle\rangle \, ,$$

$$Y_3 \, x^{o\to o} \, y^{o\to o} \to y \, (x \, U_0) \, , \qquad S_2 \, x^{o\to o} \, z^{o\to o} \, y_1^o \to x \, y_1 \, , \qquad U_1 \, z^o \to e\langle\rangle \, .$$

It is easy to see that $\mathcal{H}$ can be computed in logarithmic space (in particular its size is polynomial in the size of $\mathcal{G}$). We also notice that the order of the scheme remains unchanged; this is the case because $ord(\mathbf{H}(\alpha)) = ord(\alpha)$ for every sort α.

It remains to prove that $BT(\Lambda(\mathcal{H})) = BT(\Lambda(\mathcal{G}))$ for every closed λ-term M^o. To this end, we need to define a variant of our transformation that works with λ-terms, not with schemes. We thus define $\mathbf{R}_k^\Lambda(M)$ is the same way as $\mathbf{R}_k(M)$, but in the definition we replace $S_{\alpha,i}$ with $\mathcal{R}'(S_{\alpha,i})$ (recall that $\mathcal{R}'$ describes rules of the new scheme). Similarly $\mathbf{L}_n^\Lambda(N)$ is defined as $\mathbf{L}_n(N)$, but in the definition we replace U_i with $\mathcal{R}'(U_i)$. Finally, $\mathbf{H}_\Omega^\Lambda(M)$ is defined as $\mathbf{H}_\Omega(M)$, but it uses functions $\mathbf{R}_i^\Lambda$ and $\mathbf{L}_i^\Lambda$ instead of $\mathbf{R}_i$ and $\mathbf{L}_i$. In other words, this variant of the transformation inserts definitions of the nonterminals $S_{\alpha,i}$ and U_i instead of the nonterminals themselves.

We immediately see that $\Lambda(\mathcal{H}) = \mathbf{H}_\emptyset^\Lambda(\Lambda(\mathcal{G}))$. In the remaining part of the section we will prove that $BT(\mathbf{H}_\emptyset^\Lambda(M)) = BT(M)$ for every closed λ-term M^o; this implies that $BT(\Lambda(\mathcal{H})) = BT(\Lambda(\mathcal{G}))$ when instantiated with $M = \Lambda(\mathcal{G})$. The proof is split to several lemmata.

▶ **Lemma 3.** *Let P be a λ-term of sort $\mathbf{R}_{k-1}(\mathbf{H}(\alpha))$, where $k > ord(\alpha)$. In such a situation $\mathcal{R}'(S_{\alpha,k}) P \mathcal{R}'(U_{k-1}) \to_{\beta\eta}^* P$.*

Proof. Let $\mathbf{R}_{k-1}(\mathbf{H}(\alpha)) = \beta_1 \to \cdots \to \beta_s \to o$. Recalling the definition of $\mathcal{R}'(S_{\alpha,k})$ we observe that

$$\mathcal{R}'(S_{\alpha,k}) P \mathcal{R}'(U_{k-1}) = (\lambda x.\lambda z.\lambda y_1.\cdots.\lambda y_s.x\, y_1\, \ldots\, y_s)\, P\, \mathcal{R}'(U_{k-1})$$
$$\to_\beta^2 \lambda y_1.\cdots.\lambda y_s.P\, y_1\, \ldots\, y_s \to_\eta^s P\,. \qquad \blacktriangleleft$$

▶ **Lemma 4.** *Let M be a λ-term of sort $\mathbf{H}(\alpha)$, and let $k \geq ord(\alpha) = n$. In such a situation $\mathbf{L}_n^\Lambda(\mathbf{R}_k^\Lambda(M))) \to_{\beta\eta}^* M$.*

Proof. The thesis follows directly from Lemma 3 once we recall that

$$\mathbf{L}_n^\Lambda(\mathbf{R}_k^\Lambda(M)) = \mathcal{R}'(S_{\alpha,k})\,(\mathcal{R}'(S_{\alpha,k-1})\,\ldots\,(\mathcal{R}'(S_{\alpha,n+1})\,M)\ldots)$$
$$\mathcal{R}'(U_{k-1})\,\mathcal{R}'(U_{k-2})\,\ldots\,\mathcal{R}'(U_n)\,. \qquad \blacktriangleleft$$

▶ **Lemma 5.** *Let M and N^α be λ-terms, x^α a variable, and Ω a raise environment such that $FV(M) \setminus \{x^\alpha\} \cup FV(N) \subseteq \mathrm{dom}(\Omega)$. Let also $\alpha' = \mathbf{R}_k(\mathbf{H}(\alpha))$ for some $k \geq ord(\alpha)$. In such a situation $\mathbf{H}_{\Omega[x^\alpha \mapsto \alpha']}^\Lambda(M)[\mathbf{R}_k^\Lambda(\mathbf{H}_\Omega^\Lambda(N))/x^{\alpha'}] \to_{\beta\eta}^\infty \mathbf{H}_\Omega^\Lambda(M[N/x^\alpha])$.*

Proof. The proof is by coinduction on the structure of M. Only the case of $M = x^\alpha$ is interesting. In this case $\mathbf{H}_{\Omega[x^\alpha \mapsto \alpha']}^\Lambda(M) = \mathbf{L}_{ord(\alpha)}^\Lambda(x^{\alpha'})$, so $\mathbf{H}_{\Omega[x^\alpha \mapsto \alpha']}^\Lambda(M)[\mathbf{R}_k^\Lambda(\mathbf{H}_\Omega^\Lambda(N))/x^{\alpha'}] = \mathbf{L}_{ord(\alpha)}^\Lambda(\mathbf{R}_k^\Lambda(\mathbf{H}_\Omega^\Lambda(N)))$, and by Lemma 4 we have that $\mathbf{L}_{ord(\alpha)}^\Lambda(\mathbf{R}_k^\Lambda(\mathbf{H}_\Omega^\Lambda(N))) \to_{\beta\eta}^* \mathbf{H}_\Omega^\Lambda(N)$, which is what we need since $M[N/x^\alpha] = N$.

We remark that in the case of $M = \lambda y^\beta.K$, we use the assumption of coinduction for the extended raise environment $\Omega[y^\beta \mapsto \beta']$, and we observe that $\mathbf{H}_\Omega^\Lambda(N) = \mathbf{H}_{\Omega[y^\beta \mapsto \beta']}^\Lambda(N)$ when (without loss of generality) we assume that y^β is not free in N. $\qquad \blacktriangleleft$

▶ **Lemma 6.** *If $M \xrightarrow{h}_\beta N$, and Ω is a raise environment such that $FV(M) \subseteq \mathrm{dom}(\Omega)$, then $(\mathbf{H}_\Omega^\Lambda(M), \mathbf{H}_\Omega^\Lambda(N)) \in (\xrightarrow{h}_\beta) \circ (\to_{\beta\eta}^\infty)$.*

Proof. The proof is by induction on the depth of the head redex in M. The induction step is trivial. Consider thus the base case, when $M = (\lambda x^\alpha.K)\, L$, and $N = K[L/x^\alpha]$. Let $k = ord(\lambda x.K) - 1$, and $\alpha' = \mathbf{R}_k(\mathbf{H}(\alpha))$; clearly $k \geq ord(\alpha)$. By definition we have that

$$\mathbf{H}_\Omega^\Lambda(M) = (\lambda x^{\alpha'}.\mathbf{H}_{\Omega[x^\alpha \mapsto \alpha']}^\Lambda(K))\, \mathbf{R}_k^\Lambda(\mathbf{H}_\Omega^\Lambda(L))\,.$$

Taking $P = \mathbf{H}^\Lambda_{\Omega[x^\alpha \mapsto \alpha']}(K)[\mathbf{R}^\Lambda_k(\mathbf{H}^\Lambda_\Omega(L))/x^{\alpha'}]$ we see that $\mathbf{H}^\Lambda_\Omega(M) \xrightarrow{h}_\beta P$, and from Lemma 5 we obtain that $P \to^\infty_{\beta\eta} \mathbf{H}^\Lambda_\Omega(N)$. ◀

Using Lemma 6 it is easy to prove by coinduction that for every closed λ-term M of sort o it holds that $BT(\mathbf{H}^\Lambda_\emptyset(M)) = BT(M)$. Let us write this in details. The proof is by coinduction on the structure of these Böhm trees. According to the definition of a Böhm tree, we have two cases. The first of them is that $M \xrightarrow{h}^*_\beta N$ for some N that starts with a node constructor. In this case, by Lemma 6 (applied to every reduction in the sequence of reductions witnessing $M \xrightarrow{h}^*_\beta N$) we have that $(\mathbf{H}^\Lambda_\emptyset(M), \mathbf{H}^\Lambda_\emptyset(N)) \in ((\xrightarrow{h}_\beta) \circ (\to^\infty_{\beta\eta}))^*$. Clearly $(\xrightarrow{h}_\beta) \subseteq (\to^\infty_{\beta\eta})$, thus using Fact 1 (multiple times) we obtain that $BT(\mathbf{H}^\Lambda_\emptyset(M)) = BT(\mathbf{H}^\Lambda_\emptyset(N))$. Let us write $N = a\langle K_1, \ldots, K_r\rangle$; then $\mathbf{H}^\Lambda_\emptyset(N) = a\langle \mathbf{H}^\Lambda_\emptyset(K_1), \ldots, \mathbf{H}^\Lambda_\emptyset(K_r)\rangle$. Since $BT(\mathbf{H}^\Lambda_\emptyset(K_i)) = BT(K_i)$ by the assumption of coinduction, we can conclude that

$$BT(\mathbf{H}^\Lambda_\emptyset(M)) = BT(\mathbf{H}^\Lambda_\emptyset(N)) = a\langle BT(\mathbf{H}^\Lambda_\emptyset(K_1)), \ldots, BT(\mathbf{H}^\Lambda_\emptyset(K_r))\rangle$$
$$= a\langle BT(K_1), \ldots, BT(K_r)\rangle = BT(M).$$

The opposite case is that no sequence of head β-reductions from M leads to a λ-term starting with a node constructor. It is then possible that $M \xrightarrow{h}^*_\beta N$ for some N such that no head β-reduction can be performed from N (but N does not start with a node constructor). Since M, and thus also N, are closed and of sort o, this implies that N is of the form $\ldots K_3 K_2 K_1$ (infinite application). From the definition of $\mathbf{H}^\Lambda_\emptyset$ it follows that $\mathbf{H}^\Lambda_\emptyset(N)$ is also such an infinite application, and thus no head β-reduction can be performed from N. Moreover, as in the previous case, we can see that $BT(\mathbf{H}^\Lambda_\emptyset(M)) = BT(\mathbf{H}^\Lambda_\emptyset(N))$. We thus have

$$BT(\mathbf{H}^\Lambda_\emptyset(M)) = BT(\mathbf{H}^\Lambda_\emptyset(N)) = \bot\langle\rangle = BT(M).$$

Another possibility is that an infinite sequence of head β-reductions can be performed from M. In other words, for every $n \in \mathbb{N}$ there is a λ-term N such that $M \xrightarrow{h}^n_\beta N$. Fix some such n and N. Lemma 6 implies that $(\mathbf{H}^\Lambda_\emptyset(M), \mathbf{H}^\Lambda_\emptyset(N)) \in ((\xrightarrow{h}_\beta) \circ (\to^\infty_{\beta\eta}))^n$. Using Fact 7 (below) we can move all head β-reductions to the front, and obtain that $(\mathbf{H}^\Lambda_\emptyset(M), \mathbf{H}^\Lambda_\emptyset(N)) \in (\xrightarrow{h}_\beta)^n \circ (\to^\infty_{\beta\eta})^n$ (we suppress the proof of Fact 7, as the fact is standard, and the proof is not difficult). This can be done for every n, which means that arbitrarily long sequences of head β-reductions start in $\mathbf{H}^\Lambda_\emptyset(M)$. Recalling that for every P there is at most one Q such that $P \xrightarrow{h}_\beta Q$, and that no head β-reduction can be performed from a λ-term starting with a node constructor, we conclude that $BT(\mathbf{H}^\Lambda_\emptyset(M)) = \bot\langle\rangle = BT(M)$.

▶ **Fact 7.** *For all λ-terms M, N of sort o, if $(M, N) \in (\to^\infty_{\beta\eta}) \circ (\xrightarrow{h}_\beta)$, then $(M, N) \in (\xrightarrow{h}_\beta) \circ (\to^\infty_{\beta\eta})$.*

4 Safe Schemes

In this section we consider safe schemes. Let us recall that we have two definitions of safety. Following Carayol and Serre [6] we use the name "safe schemes" for schemes that are safe according to the modern definition, and "Damm-safe schemes" for schemes that are safe according to the definition of Damm. We now give these definitions.

We define by coinduction when an applicative term is *safe*, with respect to a set of nonterminals $\mathcal{N}$:

- $M = a\langle K_1, \ldots, K_r\rangle$ is safe if $K_1, \ldots, K_r$ are safe,
- $M = x \in \mathcal{V}$ (in particular $M = X \in \mathcal{N}$) is always safe, and
- $M = K\, L_1 \ldots L_s$ (with $s \geq 1$) is safe if $K, L_1, \ldots, L_s$ are safe, and additionally $ord(x) \geq ord(M)$ for all $x \in FV(M) \setminus \mathcal{N}$.

Damm-safe applicative terms are also defined by coinduction:

- $M = a\langle K_1, \ldots, K_r\rangle$ is Damm-safe if $K_1, \ldots, K_r$ are Damm-safe,
- $M = x \in \mathcal{V}$ (in particular $M = X \in \mathcal{N}$) is always Damm-safe, and
- $M = K\, L_1 \ldots L_s$ (with $s \geq 1$) is Damm-safe if $K, L_1, \ldots, L_s$ are Damm-safe, and additionally $ord(L_i) \geq ord(M)$ for all $i \in \{1, \ldots, s\}$.

A scheme $\mathcal{G} = (\mathcal{N}, \mathcal{R}, X_0)$ is safe (Damm-safe) if the right side of every of its rules (i.e., the term M when $\mathcal{R}(X) = \lambda x_1.\cdots.\lambda x_s.M$) is safe (Damm-safe, respectively).

Notice that not every subterm of a (Damm-)safe term need to be (Damm-)safe. But, for example, subterms appearing as arguments are (Damm-)safe, etc. We remark that the definition of safe applicative terms can be extended to λ-terms which are not applicative [3], but we refrain from doing this.

▶ **Example 2.** Consider a scheme $\mathcal{G}_2$ with the following rules:

$$S \to W\,(X\,(b\langle\rangle))\,, \qquad W\,g^{(o\to o)\to o} \to Y\,(X\,(Y\,g))\,, \qquad Y\,g^{(o\to o)\to o} \to g\,A\,,$$
$$X\,x^o\,f^{o\to o} \to f\,x\,, \qquad A\,x^o \to a\langle x\rangle\,.$$

This scheme is safe, but not Damm-safe; in particular the subterm $X\,(Y\,g)$ is not Damm-safe since $ord(Y\,g) = 0 < 2 = ord(X\,(Y\,g))$. Moreover, the sort of X is not homogeneous. Notice that $BT(\Lambda(\mathcal{G}_2)) = a\langle a\langle b\langle\rangle\rangle\rangle$.

It is easy to prove by coinduction that every Damm-safe applicative term is also safe; in consequence every Damm-safe scheme is also safe. We now give two transformations: first we show how to convert a safe scheme into an equivalent scheme that is Damm-safe; then we show how to convert a Damm-safe scheme into an equivalent scheme that is Damm-safe and homogeneous.

▶ **Theorem 8.** *For every safe scheme $\mathcal{G} = (\mathcal{N}, \mathcal{R}, X_0)$ one can construct in logarithmic space a Damm-safe scheme $\mathcal{H} = (\mathcal{N}', \mathcal{R}', Y_0)$ that is of the same order as $\mathcal{G}$ and such that $BT(\Lambda(\mathcal{H})) = BT(\Lambda(\mathcal{G}))$.*

Let us fix some (arbitrary) order $\prec$ on variables. When $FV(M) \setminus \{\mathcal{N}\} = \{x_1, \ldots, x_k\}$, where $x_1 \prec \cdots \prec x_k$, then we write $OV(M)$ for the sequence $(x_1, \ldots, x_k)$.

The transformation of Theorem 8 amounts to splitting every rule of $\mathcal{G}$ into multiple simpler rules. More precisely, for every safe subterm M of the right side of every rule of $\mathcal{G}$, and for every nonterminal $M = X \in \mathcal{N}$, we create a new nonterminal denoted $\lfloor M\rfloor$. If $OV(M) = (x_1^{\alpha_1}, \ldots, x_k^{\alpha_k})$, and if the sort of M is β, then the sort of $\lfloor M\rfloor$ is $\alpha_1 \to \cdots \to \alpha_k \to \beta$. To the new set of nonterminals $\mathcal{N}'$, we take all such nonterminals $\lfloor M\rfloor$. As the starting nonterminal we take $Y_0 = \lfloor X_0\rfloor$.

We now define $\mathcal{R}'(\lfloor M\rfloor)$ for every nonterminal $\lfloor M\rfloor \in \mathcal{N}'$. Consider first the case when $M = X$ is a nonterminal from $\mathcal{N}$. Suppose that $\mathcal{R}(X) = \lambda x_1.\cdots.\lambda x_s.K$, and $OV(K) = (y_1, \ldots, y_k)$. In such a situation we put $\mathcal{R}'(\lfloor M\rfloor) = \lambda x_1.\cdots.\lambda x_s.\lfloor K\rfloor\,y_1\,\ldots\,y_r$ (on the list $y_1, \ldots, y_r$ we have those of $x_1, \ldots, x_s$ which are free in K, reordered according to $\prec$).

Suppose now that M is not a nonterminal from $\mathcal{N}$. Let $OV(M) = (x_1, \ldots, x_k)$. Let also $y_1, \ldots, y_s$ be variables of sorts $\alpha_1, \ldots, \alpha_s$, where $\alpha_1 \to \cdots \to \alpha_s \to o$ is the sort of M. We have three possibilities, depending on the shape of M.

- If $M = a\langle K_1, \ldots, K_r \rangle$, and $OV(K_i) = (z_{i,1}, \ldots, z_{i,m_i})$ for all $i \in \{1, \ldots, r\}$, then

$$\mathcal{R}'(\lfloor M \rfloor) = \lambda x_1. \cdots . \lambda x_k . a\langle \lfloor K_1 \rfloor z_{1,1} \ldots z_{1,m_1}, \ldots, \lfloor K_r \rfloor z_{r,1} \ldots z_{r,m_r} \rangle.$$

- If $M = x$, then $\mathcal{R}'(\lfloor M \rfloor) = \lambda x.\lambda y_1. \cdots . \lambda y_s . x \, y_1 \ldots y_s$.
- If $M = K_0 K_1 \ldots K_r$, where $r \geq 1$, and K_0 is not an application, and $OV(K_i) = (z_{i,1} \ldots z_{i,m_i})$ for all $i \in \{0, \ldots, r\}$, then

$$\mathcal{R}'(\lfloor M \rfloor) = \lambda x_1. \cdots . \lambda x_k . \lambda y_1. \cdots . \lambda y_s . \lfloor K_0 \rfloor z_{0,1} \ldots z_{0,m_0}$$
$$(\lfloor K_1 \rfloor z_{1,1} \ldots z_{1,m_1}) \ldots (\lfloor K_r \rfloor z_{r,1} \ldots z_{r,m_r}) \, y_1 \ldots y_s.$$

Notice that in the first and the third case, the subterms K_i are safe, so $\lfloor K_i \rfloor$ is indeed a nonterminal in $\mathcal{N}'$. It is also easy to prove that the right side of every rule is Damm-safe. Indeed, for subterms of sort o (i.e., of order 0) there is nothing to check. The only subterms which are of higher order (and which are not a part of a larger application) are $\lfloor K_i \rfloor z_{i,1} \ldots z_{i,m_i}$ in the last case of the definition. By safety of K_i we have that $ord(z_{i,j}) \geq ord(K_i)$, since $z_{i,j}$ is free in K_i, and exactly this is needed to claim that $\lfloor K_i \rfloor z_{i,1} \ldots z_{i,m_i}$ is Damm-safe.

Let $Exp(K)$ be the λ-term obtained by repeatedly replacing in K all nonterminals $\lfloor L \rfloor$ such that $L \notin \mathcal{N}$ by $\mathcal{R}'(L)$ (this is similar to $\Lambda_{\mathcal{H}}(K)$, but we do not expand nonterminals of the form $\lfloor X \rfloor$, where $X \in \mathcal{N}$). It is easy to prove by induction on the structure of a finite applicative term M, that if $OV(M) = (x_1, \ldots, x_k)$, then $Exp(\mathcal{R}'(\lfloor M \rfloor)) \, x_1 \ldots x_k \to_{\beta\eta}^* M$ (if we identify nonterminals $X \in \mathcal{N}$ with $\lfloor X \rfloor$). In consequence $\Lambda(\mathcal{H}) \to_{\beta\eta}^\infty \Lambda(\mathcal{G})$, which implies that $BT(\Lambda(\mathcal{H})) = BT(\Lambda(\mathcal{G}))$, by Fact 1.

▶ **Example 2** (continued). While applying our transformation to the safe scheme $\mathcal{G}_2$ from Example 2, we obtain a Damm-safe scheme $\mathcal{H}_2$ with the following rules (where variables x, f, g are of sorts o, $o \to o$, and $(o \to o) \to o$, respectively; we assume that $f \prec g \prec x$):

$$\lfloor S \rfloor \to \lfloor W \, (X \, (b\langle\rangle)) \rfloor, \qquad \lfloor X \rfloor x \, f \to \lfloor f \, x \rfloor f \, x, \qquad \lfloor g \rfloor g \, f \to g \, f,$$
$$\lfloor W \rfloor g \to \lfloor Y \, (X \, (Y \, g)) \rfloor g, \qquad \lfloor A \rfloor x \to \lfloor a\langle x \rangle \rfloor x, \qquad \lfloor x \rfloor x \to x,$$
$$\lfloor Y \rfloor g \to \lfloor g \, A \rfloor g, \qquad \lfloor f \rfloor f \, x \to f \, x, \qquad \lfloor b\langle\rangle \rfloor \to b\langle\rangle,$$
$$\lfloor W \, (X \, (b\langle\rangle)) \rfloor \to \lfloor W \rfloor \lfloor X \, (b\langle\rangle) \rfloor, \qquad \lfloor Y \, g \rfloor g \to \lfloor Y \rfloor (\lfloor g \rfloor g),$$
$$\lfloor X \, (b\langle\rangle) \rfloor f \to \lfloor X \rfloor \lfloor b\langle\rangle \rfloor f, \qquad \lfloor g \, A \rfloor g \to \lfloor g \rfloor g \lfloor A \rfloor,$$
$$\lfloor Y \, (X \, (Y \, g)) \rfloor g \to \lfloor Y \rfloor (\lfloor X \, (Y \, g) \rfloor g), \qquad \lfloor f \, x \rfloor f \, x \to \lfloor f \rfloor f \, (\lfloor x \rfloor x),$$
$$\lfloor X \, (Y \, g) \rfloor g \, f \to \lfloor X \rfloor (\lfloor Y \, g \rfloor g) f, \qquad \lfloor a\langle x \rangle \rfloor x \to a\langle \lfloor x \rfloor x \rangle.$$

We now come to the second transformation.

▶ **Theorem 9.** *For every Damm-safe scheme $\mathcal{G} = (\mathcal{N}, \mathcal{R}, X_0)$ one can construct in logarithmic space a homogeneous Damm-safe scheme $\mathcal{H} = (\mathcal{N}', \mathcal{R}', X_0)$ that is of the same order as $\mathcal{G}$ and such that $BT(\Lambda(\mathcal{H})) = BT(\Lambda(\mathcal{G}))$.*

We remark that the transformation from the previous section (which converts a scheme to a homogeneous scheme), when applied to a Damm-safe scheme results in a scheme that is homogeneous, but no longer (Damm-)safe. Indeed, we have there (on argument positions) subterms of the form $S_{\alpha,k+1} M$, where $k = ord(M)$. Recalling that the order of $S_{\alpha,k+1} M$ is $k+1$, we notice that such a subterm is not Damm-safe (and if, e.g., M is a variable, it is also not safe).

We thus use a different approach: we reorder parameters / arguments. This approach works only because the scheme is Damm-safe. Indeed, Damm-safety ensures that when an argument of some order k is applied, then simultaneously all arguments of orders higher than k are applied, and thus we can move our argument of order k behind these arguments.

Before giving a formal definition of our transformation, let us extend the notion of Damm-safety from applicative terms to λ-terms. To this end, to the definition of a Damm-safe terms, we add an item saying that a λ-term $M = \lambda x_1. \cdots .\lambda x_s.K$ (with $s \geq 1$) is Damm-safe if K is Damm-safe, and additionally $ord(x_i) \geq ord(K)$ for all $i \in \{1, \ldots, s\}$.

For sorts $\alpha_1, \ldots, \alpha_s$, let $sort(\alpha_1, \ldots, \alpha_s)$ be the permutation $(i_1, \ldots, i_s)$ of $(1, \ldots, s)$ for which either $ord(\alpha_{i_j}) = ord(\alpha_{i_{j+1}})$ and $i_j < i_{j+1}$, or $ord(\alpha_{i_j}) > ord(\alpha_{i_{j+1}})$, for every $j \in \{1, \ldots, s\}$. Having the sorting function, we define our transformation on sorts, by induction: when $\alpha = \alpha_1 \to \cdots \to \alpha_s \to o$, and $sort(\alpha_1, \ldots, \alpha_s) = (i_1, \ldots, i_s)$, we put $\mathbf{S}(\alpha) = \mathbf{S}(\alpha_{i_1}) \to \cdots \to \mathbf{S}(\alpha_{i_s}) \to o$ (in particular $\mathbf{S}(o) = o$). Similarly, for Damm-safe λ-terms we define by coinduction:

- if $M = a\langle K_1, \ldots, K_r \rangle$, then $\mathbf{S}(M) = a\langle \mathbf{S}(K_1), \ldots, \mathbf{S}(K_r) \rangle$,
- if $M = x^\alpha \in \mathcal{V}$, then $\mathbf{S}(M) = x^{\mathbf{S}(\alpha)}$ (where x is either a "real" variable, or a nonterminal),
- if $M = K\, L_1^{\alpha_1} \ldots L_s^{\alpha_s}$ (with $s \geq 1$), and $sort(\alpha_1, \ldots, \alpha_s) = (i_1, \ldots, i_s)$, and K is Damm-safe, then $\mathbf{S}(M) = \mathbf{S}(K)\, \mathbf{S}(L_{i_1}) \ldots \mathbf{S}(L_{i_s})$, and
- finally, if $M = \lambda x_1^{\alpha_1}. \cdots .\lambda x_s^{\alpha_s}.K$ (with $s \geq 1$), and $sort(\alpha_1, \ldots, \alpha_s) = (i_1, \ldots, i_s)$, and $ord(x_i) \geq ord(K)$ for all $i \in \{1, \ldots, s\}$, then $\mathbf{S}(M) = \lambda x_{i_1}^{\mathbf{S}(\alpha_{i_1})}. \cdots .\lambda x_{i_s}^{\mathbf{S}(\alpha_{i_s})}.\mathbf{S}(K)$.

Notice that for a λ-term M of sort α, the sort of $\mathbf{S}(M)$ is $\mathbf{S}(\alpha)$.

It may appear that the definition is ambiguous (but it is not). The problem is that while transforming an application $M = K\, L_1 \ldots L_{k+m}$, where both K and $N = K\, L_1 \ldots L_k$ are Damm-safe, we may proceed in two ways: we may sort all the arguments $L_1 \ldots L_{k+m}$, but we may also separately sort the arguments $L_1 \ldots L_k$ and separately the arguments $L_{k+1} \ldots L_{k+m}$. We notice, though, that the effect will be the same. Indeed, we have that $ord(L_i) \geq ord(N)$ for $i \leq k$, because N is Damm-safe, and $ord(L_i) < ord(N)$ for $i > k$ because these L_i are given as arguments to N. This means that even while sorting all the arguments $L_1 \ldots L_{k+m}$ together, the arguments L_i for $i \leq k$ will appear before the arguments for $i > k$. The same can be said about a sequence of λ-binders $M = \lambda x_1. \cdots .\lambda x_{k+m}.K$ in which $ord(x_i) \geq ord(\lambda x_{k+1}. \cdots .\lambda x_{k+m}.K)$ for all $i \in \{1, \ldots, k\}$.

Having a transformation of λ-terms, it is immediate to define a transformation on schemes: we take $\mathcal{N}' = \{X^{\mathbf{S}(\alpha)} \mid X^\alpha \in \mathcal{N}\}$, and $\mathcal{R}'(X^{\mathbf{S}(\alpha)}) = \mathbf{S}(\mathcal{R}(X^\alpha))$ for all $X^\alpha \in \mathcal{N}$.

On the one hand, it should be clear that $\mathcal{H}$ is homogeneous, Damm-safe, and of the same order as $\mathcal{G}$. On the other hand, it is easy to prove the following lemma.

▶ **Lemma 10.** *If* $M = (\lambda x_1. \cdots .\lambda x_s.K)\, L_1 \ldots L_s$ *is a Damm-safe λ-term, and* $M \xrightarrow{h}{}_\beta^s N$, *then N is Damm-safe, and* $\mathbf{S}(M) \xrightarrow{h}{}_\beta^s \mathbf{S}(N)$.

Using the above lemma it is easy to prove by coinduction that $BT(\mathbf{S}(M)) = BT(M)$ for every Damm-safe λ-term M. Because $\Lambda(\mathcal{H}) = \mathbf{S}(\Lambda(\mathcal{G}))$, and because $\Lambda(\mathcal{G})$ is Damm-safe, it follows that $BT(\Lambda(\mathcal{H})) = BT(\Lambda(\mathcal{G}))$. Notice that in Lemma 10 it is essential that we perform all the s head β-reductions at once, not only a single one (since in $\mathbf{S}(M)$ the s arguments are applied in different order than in M).

▶ **Example 2** (continued). Let us apply the transformation to the Damm-safe scheme $\mathcal{H}_2$ from our example. Since $\lfloor X \rfloor$ is the only nonterminal having a non-homogeneous sort, only

the rules involving $\lfloor X \rfloor$ are changed, as follows:

$$\lfloor X \rfloor \, f \, x \to \lfloor f \, x \rfloor \, f \, x \,, \qquad\qquad \lfloor X \, (Y \, g) \rfloor \, g \, f \to \lfloor X \rfloor \, f \, (\lfloor Y \, g \rfloor \, g) \,,$$
$$\lfloor X \, (b\langle\rangle) \rfloor \, f \to \lfloor X \rfloor \, f \, \lfloor b\langle\rangle \rfloor \,.$$

Notice that it does not make sense to apply the transformation to the scheme $\mathcal{G}_2$, which is not Damm-safe. Indeed, it would be impossible to swap the order of the parameters of X, since in the subterm $X \, (Y \, g)$ we are applying only one argument to X.

5 Consequences of Homogeneity

Let us say that a λ-term is homogeneous if sorts of all its subterms are homogeneous. By definition this means that arguments of higher order are always applied before arguments of lower order. Due to this fact, in a homogeneous λ-term (unlike in an arbitrary λ-term) we can perform β-reductions starting from redexes concerning variables of the highest order. In this section we formalize and prove this property of homogeneous λ-terms (Lemmata 11 and 12). We remark that this property turned out to be useful e.g. in Parys [17].

We define the order of a β-reduction as the order of the involved variable. More precisely, for a number $k \in \mathbb{N}$, the relation $\to_{\beta(k)}$ of β-reduction of order k is defined as the compatible closure of the relation $\{((\lambda x.K) \, L, K[L/x]) \mid ord(x) = k\}$.

We first give our result for finite λ-terms.

▶ **Lemma 11.** *Let M be a finite closed homogeneous λ-term of sort o and complexity at most n. Then there exist λ-terms $N_n, N_{n-1}, \ldots, N_0$ such that $M = N_n$, and for every $k \in \{0, \ldots, n-1\}$, N_k is of complexity at most k and $N_{k+1} \to^*_{\beta(k)} N_k$, and $N_0 = BT(M)$.*

For infinite λ-terms we need to be slightly more careful: it is not enough to replace the reflexive transitive closure $\to^*_{\beta(k)}$ by the infinitary closure $\to^\infty_{\beta(k)}$. The problem lies in subterms which do not have so-called head normal form: infinite applications $\ldots K_3 \, K_2 \, K_1$, and subterms from which we can perform infinitely many head β-reductions. These are subterms responsible for creating nodes labeled by $\bot$ in the Böhm tree. We cannot deal with these subterms by only applying β-reductions. We need to introduce relations that explicitly replace such "invalid" subterms by $\bot\langle\rangle$.

The relation $\xrightarrow{h}_{\beta(k)}$ of *head β-reduction of order k* (where $k \in \mathbb{N}$) is defined as

$$\{((\lambda x.K) \, L \, P_1 \, \ldots \, P_n, K[L/x] \, P_1 \, \ldots \, P_n) \mid ord(x) = k\} \,.$$

Consider now the relation containing all pairs of the form $(K, \lambda x_1. \cdots . \lambda x_s. \bot\langle\rangle)$, where K and $\lambda x_1. \cdots . \lambda x_s. \bot\langle\rangle$ are of the same sort, and either for every $n \in \mathbb{N}$ there is L such that $K \xrightarrow{h}^n_{\beta(k)} L$, or K is an infinite application. The compatible closure of this relation is denoted $\to_{\bot(k)}$. By $\to_{\beta\bot(k)}$ we denote the union of $\to_{\beta(k)}$ and $\to_{\bot(k)}$. Using this relation we can now reformulate Lemma 11 for infinite λ-terms.

▶ **Lemma 12.** *Let M be a closed homogeneous λ-term of sort o and complexity at most n. Then there exist λ-terms $N_n, N_{n-1}, \ldots, N_0$ such that $M = N_n$, and for every $k \in \{0, \ldots, n-1\}$, N_k is of complexity at most k and $N_{k+1} \to^\infty_{\beta\bot(k)} N_k$, and $N_0 = BT(M)$.*

Notice that Lemma 11 is an immediate consequence of Lemma 12, because when a λ-term K is finite, then there is no L such that $K \to_{\bot(k)} L$, and $K \to^\infty_{\beta(k)} M$ implies $K \to^*_{\beta(k)} M$ (every sequence of β-reductions from a finite λ-term is finite). Lemma 12, in turn, is a consequence of the following lemma.

▶ **Lemma 13.** *Let M be a λ-term of complexity at most k, order at most $k-1$, and such that all free variables of M have order at most $k-1$. Then there exists a λ-term P of complexity at most $k-1$ such that $M \to_{\beta\perp(k)}^{\infty} P$.*

Proof. The proof is by coinduction. Suppose first that for every $n \in \mathbb{N}$ there is N such that $M \xrightarrow{h}{}_{\beta(k)}^{n} N$. In this situation $M \to_{\perp(k)} \lambda x_1. \cdots . \lambda x_s. \perp \langle \rangle$ (for an appropriate sequence of variables $x_1, \ldots, x_s$, corresponding to the sort of M). Denoting the latter λ-term P we obtain the thesis, since the complexity of P equals $ord(P) = ord(M) \leq k-1$.

The opposite case is that $M \xrightarrow{h}{}_{\beta(k)}^{*} N$ for some N, but there is no N' such that $N \xrightarrow{h}{}_{\beta(k)} N'$. When N is a variable, the thesis is trivial for $P = N$, and when $N = a \langle K_1, \ldots, K_r \rangle$, the thesis follows directly from the assumption of coinduction. When $N = \lambda x.K$, the thesis also follows from the assumption of coinduction; we only need to observe that $ord(N) = ord(M) \leq k-1$ implies that $ord(x) \leq k-2 \leq k-1$. Suppose thus that N is an application. When N is an infinite application, we again have $M \to_{\perp(k)} \lambda x_1. \cdots . \lambda x_s. \perp \langle \rangle$, and we are done. When $N = x\, L_1 \ldots L_s$, by assumption the order of x is at most $k-1$, so we can simply use the assumption of coinduction for all L_i. Otherwise N is of the form $(\lambda x.K)\, L_1 \ldots L_s$. Since no head β-reduction of order k starts in N, necessarily $ord(x) \neq k$. Knowing that the complexity of N is at most k, and that the sort of $\lambda x.K$ is homogeneous, this implies that $ord(\lambda x.K) = ord(x) - 1 \leq k-1$. We can thus again use the assumption of coinduction for all the subterms $K, L_1, \ldots, L_s$. ◀

▶ **Remark.** We notice that Lemmata 11 and 12 would be false if we have allowed λ-terms involving non-homogeneous sorts. For example, from a λ-term of the form $(\lambda x.\lambda y.K)\, L\, M$ with $ord(x) = 0$ and $ord(y) = 1$ we have to perform a β-reduction of order 0 concerning x before a β-reduction of order 1 concerning y. It is, though, possible to reformulate our lemmata without the homogeneity assumption. One only has to define the order of a β-reduction $(\lambda x.K)\, L \to_\beta K[L/x]$ in a less natural way, as $ord(\lambda x.K) - 1$, not as $ord(x)$ (notice that these two numbers coincide for homogeneous sorts).

───── **References** ─────

1   Alfred V. Aho. Indexed grammars - an extension of context-free grammars. *J. ACM*, 15(4):647–671, 1968. `doi:10.1145/321479.321488`.

2   William Blum. Type homogeneity is not a restriction for safe recursion schemes. *CoRR*, abs/1701.02118, 2017. `arXiv:1701.02118`.

3   William Blum and C.-H. Luke Ong. The safe lambda calculus. *Logical Methods in Computer Science*, 5(1), 2009. `doi:10.2168/LMCS-5(1:3)2009`.

4   Luca Breveglieri, Alessandra Cherubini, Claudio Citrini, and Stefano Crespi-Reghizzi. Multi-push-down languages and grammars. *Int. J. Found. Comput. Sci.*, 7(3):253–292, 1996. `doi:10.1142/S0129054196000191`.

5   Christopher H. Broadbent. *On collapsible pushdown automata, their graphs and the power of links*. PhD thesis, University of Oxford, 2011. URL: `https://ora.ox.ac.uk/objects/uuid:aaa328fe-60be-4abe-8f84-97dbd9e0a3fe`.

6   Arnaud Carayol and Olivier Serre. Collapsible pushdown automata and labeled recursion schemes: Equivalence, safety and effective selection. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, pages 165–174, 2012. `doi:10.1109/LICS.2012.73`.

7   Lorenzo Clemente, Paweł Parys, Sylvain Salvati, and Igor Walukiewicz. Ordered tree-pushdown systems. In *35th IARCS Annual Conference on Foundation of Software Techno-*

logy and Theoretical Computer Science, FSTTCS 2015, December 16-18, 2015, Bangalore, India, pages 163–177, 2015. `doi:10.4230/LIPIcs.FSTTCS.2015.163`.

8 Łukasz Czajka. Coinductive techniques in infinitary lambda-calculus. *CoRR*, abs/1501.04354, 2015. `arXiv:1501.04354`.

9 Werner Damm. The IO- and OI-hierarchies. *Theor. Comput. Sci.*, 20:95–207, 1982. `doi:10.1016/0304-3975(82)90009-3`.

10 Matthew Hague, Andrzej S. Murawski, C.-H. Luke Ong, and Olivier Serre. Collapsible pushdown automata and recursion schemes. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*, pages 452–461, 2008. `doi:10.1109/LICS.2008.34`.

11 Richard Kennaway, Jan Willem Klop, M. Ronan Sleep, and Fer-Jan de Vries. Infinitary lambda calculus. *Theor. Comput. Sci.*, 175(1):93–125, 1997. `doi:10.1016/S0304-3975(96)00171-5`.

12 Richard Kennaway, Vincent van Oostrom, and Fer-Jan de Vries. Meaningless terms in rewriting. *Journal of Functional and Logic Programming*, 1999(1), 1999. URL: `http://danae.uni-muenster.de/lehre/kuchen/JFLP/articles/1999/A99-01/A99-01.html`.

13 Teodor Knapik, Damian Niwinski, and Paweł Urzyczyn. Deciding monadic theories of hyperalgebraic trees. In *TLCA*, pages 253–267, 2001. `doi:10.1007/3-540-45413-6_21`.

14 Teodor Knapik, Damian Niwiński, and Paweł Urzyczyn. Higher-order pushdown trees are easy. In *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings*, pages 205–222, 2002. `doi:10.1007/3-540-45931-6\_15`.

15 Gregory M. Kobele and Sylvain Salvati. The IO and OI hierarchies revisited. *Inf. Comput.*, 243:205–221, 2015. `doi:10.1016/j.ic.2014.12.015`.

16 Paweł Parys. On the significance of the collapse operation. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, pages 521–530, 2012. `doi:10.1109/LICS.2012.62`.

17 Paweł Parys. The complexity of the diagonal problem for recursion schemes. In *37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2017, December 11-15, 2017, Kanpur, India*, pages 45:1–45:14, 2017. `doi:10.4230/LIPIcs.FSTTCS.2017.45`.

18 Sylvain Salvati and Igor Walukiewicz. Simply typed fixpoint calculus and collapsible pushdown automata. *Mathematical Structures in Computer Science*, 26(7):1304–1350, 2016. `doi:10.1017/S0960129514000590`.

Nominal Unification with Atom and Context Variables

Manfred Schmidt-Schauß[1]

Goethe-University Frankfurt, Germany
schauss@ki.cs.uni-frankfurt.de
  https://orcid.org/0000-0001-8809-7385

David Sabel[2]

Goethe-University Frankfurt, Germany
sabel@ki.cs.uni-frankfurt.de
  https://orcid.org/0000-0002-5109-3273

─── **Abstract** ───

Automated deduction in higher-order program calculi, where properties of transformation rules are demanded, or confluence or other equational properties are requested, can often be done by syntactically computing overlaps (critical pairs) of reduction rules and transformation rules. Since higher-order calculi have alpha-equivalence as fundamental equivalence, the reasoning procedure must deal with it. We define ASD1-unification problems, which are higher-order equational unification problems employing variables for atoms, expressions and contexts, with additional distinct-variable constraints, and which have to be solved w.r.t. alpha-equivalence. Our proposal is to extend nominal unification to solve these unification problems. We succeeded in constructing the nominal unification algorithm NomUnifyASD. We show that NomUnifyASD is sound and complete for this problem class, and outputs a set of unifiers with constraints in nondeterministic polynomial time if the final constraints are satisfiable. We also show that solvability of the output constraints can be decided in NEXPTIME, and for a fixed number of context-variables in NP time. For terms without context-variables and atom-variables, NomUnifyASD runs in polynomial time, is unitary, and extends the classical problem by permitting distinct-variable constraints.

1 Introduction

Automated deduction in higher-order program calculi, where properties of transformation rules are demanded, or confluence or other equational properties are requested, can often be done by syntactically computing overlaps (critical pairs) of reduction rules and transformation rules. Since higher-order calculi have alpha-equivalence as fundamental equivalence, the reasoning procedure must deal with it. We define ASD1-unification problems, which are higher-order equational unification problems with variables for atoms, expressions, and contexts, with

---

additional distinct-variable constraints and which have to be solved w.r.t. alpha-equivalence. Our proposal is to extend nominal unification to solve these unification problems. The appeal of classical nominal unification is that it solves higher-order equations modulo α-equivalence in quadratic time and outputs at most a single most general unifier [37, 6, 17].

Our intended application is the *diagram-method*, which is a syntactic proof method (e.g. [38, 12, 34]) to show properties like correctness of program transformations. As an example consider the reduction rule (cp) in the call-by-need lambda-calculus with let (see e.g. [2, 23, 34]) $(\mathtt{let}\ x = \lambda y.S\ \mathtt{in}\ C[x]) \to (\mathtt{let}\ x = \lambda y.S\ \mathtt{in}\ C[\lambda y.S])$ with the restriction that x is not bound by context C. The diagram-based proof method for correctness of program transformations computes possible overlaps of the left-hand side (and in certain cases also of the right-hand side) of a program transformation with the left-hand sides of the reduction rules. An example equation is $\mathtt{let}\ A_1 = \lambda A_2.S_1\ \mathtt{in}\ D_1[\lambda A_2.S_1] \doteq \mathtt{let}\ A_3 = \lambda A_4.S_2\ \mathtt{in}\ D_2[A_3]$ where A, S, D are variables standing for concrete variables (called atoms), expressions, and contexts, respectively. The equation comes together with constraints on possible occurrences of atoms, which can be formulated using the distinct-variable condition (DVC) [5, 2] which in turn requires to add a further renaming to the rule to fulfill it (see Example 2.5 for details).

A generalized situation for overlap computation is represented by the equation $R[\ell] = C[\ell']$, where R is a reduction context (in which reduction takes place), ℓ is a left hand side of a reduction rule, C is an arbitrary context, and ℓ' is the left hand side of a transformation rule. Provided R, C are variables for reduction contexts and general contexts, respectively, solving the unification equation $R[\ell] \doteq C[\ell']$ can be attacked by using nominal unification in the extended language. However, for the general unification problem without any restrictions we did not find an algorithm to solve it, since it seems to be too hard (we discuss the problems in Sect. 4). As a consequence, we consider a subproblem and thus our presented algorithm solves nominal unifcation problems that are restricted to be linear w.r.t. context variables, so-called permutation variables do not occur in the input, and we require DVC-constraints for all equations in the input. From the perspective of applications to reasoning in calculi, these restrictions are (almost) optimal, since linearity of contexts in general holds, the DVC is also an often used assumption, and permutation variables are not required.

Results. A sound and complete algorithm NomUnifyASD for nominal unification of ASD1-unification problems is constructed. The algorithm NomUnifyASD computes in NP time a solution including a constraint (Theorem 5.8) and the collecting version produces at most exponentially many outputs. The algorithm NomFreshASD that checks satisfiability of the constraints of a solution runs in NEXPTIME (Proposition 5.7), hence solvability of ASD1-unification problems can be decided in NEXPTIME (Theorem 5.9). Since the number of context-variables is the only parameter in the exponent of the complexity, we obtain that if the number of context-variables is fixed, then the algorithm NomUnifyASD runs as a decision algorithm in NP time.

For computing diagrams (in the diagram method), it is important to obtain a complete set of unifiers. We expect that exponentiality of the number of unifiers is not a problem, since the input is usually very small. For example, the rules of the let-calculus [2] need only one context variable. We show that DVCs are a proper generalisation of freshness constraints if combined with solving equations (Proposition 3.6). A technical innovation is that decomposition for lambda-bindings can be extended to an arbitrary number of nested lambda-bindings thanks to the DVC (see Remark 3.2, Proposition 3.3, and Example 3.4).

A corollary is that classical nominal unification (with expression-variables only) is generalized by replacing freshness constraints by DVC-constraints such that unitarity and polynomial complexity still hold (Theorem 6.1).

Previous and Related Work. Nominal techniques [27, 26] support machine-oriented reasoning on the syntactic level supporting alpha-equivalence. Nominal unification of (syntactically presented) lambda-expressions was successfully attacked and a quadratic algorithm was developed [37, 6, 17] where technical innovations are the use of permutations on the abstract level and of freshness constraints. The approach is used in higher-order logic programming [8], and in automated theorem provers like nominal Isabelle [35, 36].

There are investigations that extend the expressive power of nominal unification problems: The restriction that bound variables are seen as atoms can be relaxed: Equivariant unification [7, 10] permits atom-variables and permutation-variables which however, appear to add too much expressive power as mentioned in [7, 10]. A restricted language (allowing atom-variables, but without permutations at all) and its nominal unification is analyzed in [16]. An investigation of nominal unification with atom-variables and a lazy-guessing algorithm is described in [33, 15]. Nominal unification for a lambda-calculus with function constants and a recursive-let is developed in [30] and shown to run in NP time. Reasoning on nominal terms in higher-order rewrite systems and narrowing as a general, but not provably efficient method for unification is described in [4]. Nominal techniques to compute overlaps w.r.t. all term positions are in [3], but there are no context-variables and reduction strategies cannot be encoded. If there are no binders, then the problem statement can be generalized to first-order terms with arbitrary occurrences of context-variables and the unification problem is in PSPACE [14].

Classical nominal unification is strongly related to higher-order pattern unification [9, 18, 22, 28, 24] which is a decidable fragment of (undecidable) higher-order unification [13] and has most general unifiers (i.e. is unitary). A slight extension to pattern unification that is unitary and decidable is described in [19]. Another line of research for reasoning with binders is the foundation of higher-order abstract syntax [25], and extensions of higher-order pattern unification as in [1], which, however, cannot adequately deal with ASD1-unification problems, the problem class of NomUnifyASD. The use of deBruijn indices [11] has some advantages in representing lambda expressions and avoids alpha-renamings, but is not appropriate for our problem, since we have to deal with free variables and with context variables that may capture variables.

A proposal to the automated computation of overlaps and diagrams is in [31] where the unifcation problems are solved w.r.t. syntactic equality. This approach permits an even higher expressiveness of the language, but the support for alpha-equivalence reasoning is missing, and hence several variants of extra constraints are necessary and further reasoning needs a technically detailed analysis of renamings [29].

Outline. Sect. 2 contains the definitions and extensions of nominal syntax. In Sect. 3 the preparations for the nominal unification algorithm are done, and in Sect. 4 the algorithm NomUnifyASD is introduced, which consists of a set of (non-deterministic) rules, and also the constraint checking algorithm NomFreshASD. In Sect. 5 the properties of the algorithms are analyzed. In Sect. 6 the special case NL_{aS} is reconsidered, and we illustrate how the unification algorithm operates on examples. We conclude in Sect. 7. Due to space constraints, some of the proofs are omitted, but given in [32].

2   Nominal Languages and Nominal Unification

Let $\mathcal{F}$ be a set of function symbols where each $f \in \mathcal{F}$ has a fixed arity $ar(f) \geq 0$, and $\mathcal{F}$ contains at least two function symbols, one of arity 0 (a constant) and one of arity ≥ 2. Let $\mathcal{A}t$ be the set of *atoms* ranged over by a, b; $\mathcal{A}$ be the set of *atom-variables* ranged over by

A, B; $\mathcal{S}$ be the set of *expression-variables* ranged over by S, T standing for expressions; $\mathcal{D}$ be the set of *context-variables* ranged over by D standing for single hole-contexts; and $\mathcal{P}$ be the set of *permutation-variables* ranged over by P standing for finite permutations on $\mathcal{A}t$, i.e. bijections π on $\mathcal{A}t$ such that their *support* $supp(\pi) = \{a \in \mathcal{A}t \mid \pi(a) \neq a\}$ is a finite set.

The ground expressions are lambda-expressions extended by function symbols, where the lambda-variables are called atoms. Contexts in the ground language are expressions over a language extended with a symbol $[\cdot]$ the hole (occurring only once), where expressions can be plugged in. The language will be enriched by symbols for atoms, expressions, contexts, and permutations, where the latter are mappings that may change the names of atoms and are represented by lists of swappings $(a\,b)$.

▶ **Definition 2.1.** The syntax of NL_{aASDP} is defined by the grammar

$$
\begin{array}{llll}
X \in \mathcal{A}\mathcal{E} & ::= & a \mid A \mid \pi{\cdot}A \\
e \in \mathcal{E} & ::= & X \mid S \mid \pi{\cdot}S \mid (f\ e_1 \ldots e_{ar(f)}) \mid \lambda X.e \mid C[e] \\
C \in \mathcal{C} & ::= & [\cdot] \mid D \mid \pi{\cdot}D \mid (f\ e_1 \ldots [\cdot] \ldots e_n) \mid \lambda X.[\cdot] \mid C_1[C_2] \\
\pi & ::= & \emptyset \mid (A_1\ A_2) \cdot \pi \mid (A\ a) \cdot \pi \mid (a\ A) \cdot \pi \mid (a_1\ a_2) \cdot \pi \mid P \cdot \pi \mid P^{-1} \cdot \pi
\end{array}
$$

with the categories $\mathcal{A}\mathcal{E}$ of atom expressions, $\mathcal{E}$ of expressions, $\mathcal{C}$ of contexts, and π of permutations. Here A, S, D, and P is an atom-variable, expression-variable, context-variable, or permutation-variable, respectively.

Nested permutations are forbidden, e.g. swappings $(\pi_1{\cdot}A_1\ \pi_2{\cdot}A_2)$ are excluded for simplicity of algorithms, and since their expressive power is already available using equations and constraints. We use positions (tree addresses) in expressions, where we ignore permutation expressions in the addressing scheme. *Sublanguages of NL_{aASDP}* are denoted by NL_M, where M is a substring of $aASDP$, and the grammar is restricted accordingly. The mainly used languages are NL_a as the ground language for the solutions, NL_{aASD} as the expression language for the input, and NL_{aASDP} as the working language inside of the unification algorithm.

A *substitution* $\sigma : NL_{aASDP} \to NL_{aASDP}$ maps atom-variables to atom expressions, expression-variables to expressions, context-variables to contexts, permutation-variables to permutations. We identify a substitution with its extension to expressions. A *ground substitution* ρ is a substitution $\rho : NL_{aASDP} \to NL_a$. We use permutation application $\cdot$ as operator and syntactic symbol, we use $^{-1}$ as a syntactic symbol in P^{-1} and operator for inversion, and we abbreviate $\emptyset{\cdot}V$ by V for a variable V. We use the following operations and simplifications

$$
\begin{array}{lll}
(\pi_1{\cdot}\pi_2)(e) \to (\pi_1{\cdot}(\pi_2{\cdot}e)) & (\pi_1{\cdot}\pi_2)^{-1} \to \pi_2^{-1}{\cdot}\pi_1^{-1} & \pi{\cdot}[\cdot] \to [\cdot] \\
\pi{\cdot}(f\ e_1 \ldots e_n) \to (f\ \pi{\cdot}e_1 \ldots \pi{\cdot}e_n) & \pi{\cdot}(\lambda X.e) \to \lambda\pi{\cdot}X.\pi{\cdot}e & \pi{\cdot}C[e] \to (\pi{\cdot}C)[\pi{\cdot}e] \\
(X_1\ X_2)^{-1} \to (X_1\ X_2) & (C_1 C_2)[e] \to C_1[C_2[e]]
\end{array}
$$

which permit standardizations: in NL_a all permutation operations can be removed; in NL_{aS}, permutations can be represented as lists of swappings of length at most $|n-1|$ where n is the number of used atoms; and in NL_{aASD}, the permutation operations only lead to suspensions of the form $\pi{\cdot}A$ and $\pi{\cdot}S$, $\pi{\cdot}D$. In all languages, permutations can be represented as a composition of lists of swappings, permutation-variables P and inverses P^{-1}, and context expressions can be simplified to the form $(\pi{\cdot}D)[e]$.

Let $tops(e)$ be the top symbol of e after simplification of permutation applications, i.e. $tops(a) = \mathtt{atom}$, $tops(\lambda X.e) = \lambda$, $tops(f\ e_1 \ldots e_n) = f$, $tops(\pi{\cdot}A) = A$, $tops(\pi{\cdot}S) = S$, and $tops((\pi{\cdot}D)[e]) = D$. For an expression e or context C in NL_a, we denote with $FA(e)$ or

$FA(C)$, resp., the set of free atoms, and with $\mathcal{A}t(e)$ or $\mathcal{A}t(C)$, resp., the set of all atoms. The set of atoms that become bound in the hole of a context C, called the *captured atoms* of C, is denoted as $CA(C)$.

In NL_a, α-equivalence $\sim_\alpha$ is the closure by reflexivity and congruence and the rule $a \notin FA(e') \wedge e \sim_\alpha (a\ b){\cdot}e' \implies \lambda a.e \sim_\alpha \lambda b.e'$. We also use α-equivalence for contexts: NL_a-contexts C_1 and C_2 are α-*equivalent*, written $C_1 \sim_\alpha C_2$, iff for all atoms a, $C_1[a] \sim_\alpha C_2[a]$ holds. E.g., $\lambda a.[\cdot] \not\sim_\alpha \lambda b.[\cdot]$, $\lambda a.\lambda b.\lambda a.[\cdot] \sim_\alpha \lambda b.\lambda b.\lambda a.[\cdot]$, but $\lambda a.\lambda b.[\cdot] \not\sim_\alpha \lambda a.\lambda a.[\cdot]$. For $C_1 \sim_\alpha C_2$, it suffices if $C_1[a] \sim_\alpha C_2[a]$ for all $a \in CA(C_1) \cup CA(C_2) \cup \{a'\}$, where a' is a fresh atom. Note that $C_1 \sim_\alpha C_2$ and $e_1 \sim_\alpha e_2$ imply $C_1[e_1] \sim_\alpha C_2[e_2]$, but the reverse is wrong: $(f\ a\ \lambda a.a) \sim_\alpha (f\ a\ \lambda b.b)$, but $(f\ a\ \lambda a.[\cdot]) \not\sim_\alpha (f\ a\ \lambda b.[\cdot])$ and $a \not\sim_\alpha b$.

We explain instantiation modulo α for correctly defining solvability under DVC-restrictions.

▶ **Definition 2.2** (Instantiation modulo α)**.** For testing solvability of equations, we assume that ground substitutions map into $NL_a/\sim_\alpha$. An equivalent method is that whenever a ground substitution ρ is applied to a variable S or D, we use an α-renamed copy of $S\rho$ or $D\rho$, respectively, where the renaming is done by fresh atoms that do not occur elsewhere (called *instantiation modulo α*), and where comparison is done modulo $\sim_\alpha$.

The following definition explains free/bound variables and the satisfiability of DVC-constraints of expressions in $NL_a/\sim_\alpha$ without using fresh atoms.

▶ **Definition 2.3.** Let e be a normalized NL_{aASDP}-expression and ρ be a ground substitution mapping into $NL_a/\sim_\alpha$, such that $e\rho$ is an $NL_a/\sim_\alpha$-expression. Then we define the bound atoms $BA(e, \rho)$, and satisfiability of the DVC of (e, ρ) as follows, where the bound atoms introduced by ρ are ignored.
1. If X is an atom a or a suspension $\pi{\cdot}A$ where A is an atom-variable then $BA(X, \rho) = \emptyset$, and the DVC is satisfied.
2. $BA(\pi{\cdot}S, \rho) = \emptyset$, and the DVC is satisfied.
3. $BA(f\ e_1 \ldots e_n, \rho) = \bigcup_{i=1}^{n} BA(e_i, \rho)$. The DVC is satisfied, if for all $i \neq j$, $BA(e_i, \rho) \cap (FA(e_j\rho) \cup BA(e_j, \rho)) = \emptyset$ and for all i, the DVC holds for (e_i, ρ).
4. $BA(\lambda X.e, \rho) = BA(e, \rho) \cup \{X\rho\}$. The DVC is satisfied, if it is satisfied for (e, ρ), and if $X\rho \notin BA(e, \rho)$.
5. $BA((\pi{\cdot}D)[e], \rho) = BA(e, \rho) \cup ((\pi)\rho){\cdot}CA(D\rho)$. The DVC is satisfied, if it is satisfied for D, i.e. $CA(D\rho) \cap FA(D\rho) = \emptyset$ as well as $(BA(e, \rho) \cap ((\pi)\rho){\cdot}(FA(D\rho) \cup CA(D\rho)) = \emptyset$, and the DVC is satisfied for (e, ρ).

Note that $BA(e, \rho) \neq BA(e\rho)$, since for $e = \lambda A.S, \rho = \{A \mapsto a, S \mapsto \lambda b.(a\ b)\}$, we have $BA(e\rho) = \{a, b\}$, but $BA(e, \rho) = \{a\}$ (where we do not distinguish between an atom and the α-equivalence class of an atom).

Nominal unification is connected with formulating and solving constraints. We use well-known freshness constraints and novel DVC-constraints.

▶ **Definition 2.4** (Freshness and DVC-Constraints)**.** *Freshness constraints* in NL_{aASDP} are of the form $a \# e$ and $A \# e$, and DVC-*constraints* in NL_{aASDP} are of the form DVC(e), where e is an NL_{aASDP}-expression.

A ground substitution ρ *satisfies* $A \# e$ iff $\rho(A) \notin FA(e\rho)$; and ρ *satisfies* $a \# e$ iff $a \notin FA(e\rho)$. The *distinct variable condition (*DVC$)$ holds for an NL_a-expression e, if all bound atoms in e are distinct, and all free atoms in e are distinct from all bound atoms in e. For a ground substitution ρ, DVC(e) is satisfied, iff (e, ρ) satisfies the DVC (Definition 2.3). This is equivalent to $e\rho$ (using instantiation modulo α) satisfying the DVC.

If a ground substitution ρ satisfies all constraints of a set of freshness and/or DVC-constraints, then we say that ρ is a *solution* of the constraint set. A set of constraints is *satisfiable* iff there is a solution.

For example, $f\,a\,\lambda b.(g\,a\,\lambda c.c)$ satisfies the DVC and $f\,b\,\lambda b.b$ violates it. With $\rho = \{S \mapsto \lambda c.c\}$ we have $(f\,(\lambda a.S)\,(\lambda b.S))\rho = f\,(\lambda a.\lambda c_1.c_1)\,(\lambda b.\lambda c_2.c_2)$ and ρ satisfies $\mathrm{DVC}(f\,(\lambda a.S)\,(\lambda b.S))$. As another example, $\rho' = \{S \mapsto \lambda c.b\}$ violates $\mathrm{DVC}(f\,(\lambda a.S)\,(\lambda b.S))$.

▶ **Example 2.5.** Consider a lambda-calculus with `let` as in [2]. A reduction rule of the corresponding calculus is `let` $x{=}\lambda y.s$ `in` $C[x] \to$ `let` $x{=}\lambda y.s$ `in` $C[\lambda y.s]$ where C is a context. We represent the expressions as $(\text{let } (\lambda A_x.D[A_x])\,(\lambda A_y.S))$ and $(\text{let } (\lambda A_x.D[\lambda A_y.S])\,(\lambda A_y.S))$. However, D must not capture the atom represented by A_x, nor free atoms from S, and A_x must not occur free in S. Both conditions can be captured by the constraint that `let` $(\lambda A_x.D[A_x])\,(\lambda A_y.S)$ and `let` $(\lambda A_x.D[\lambda A_y.S])\,(\lambda A_y.S)$ have to satisfy the DVC. However, the latter violates the DVC in every case due to the two occurrences of the binder A_y. Hence, we add a renaming to the rule and represent it as `let` $(\lambda A_x.D[A_x])\,(\lambda A_y.S) \to$ `let` $(\lambda A_x.D[\lambda A_z.(A_y\ A_z)\cdot S])\,(\lambda A_y.S)$ and $A_z \# S$. Now the DVC-constraints for both expressions make sense and produce the correct conditions.

▶ **Definition 2.6.** Let L be a sublanguage of NL_{aASDP}. A *nominal unification problem* in L is a pair (Γ, ∇) where Γ is a finite set of equations $e \doteq e'$ with $e, e' \in L$ and ∇ is a finite set of freshness and DVC-constraints, where all expressions are in L. A ground substitution ρ is a *solution* of (Γ, ∇) iff ρ satisfies ∇ and $e\rho \sim_\alpha e'\rho$ for all $e \doteq e' \in \Gamma$. A *unifier* for (Γ, ∇) is a pair (σ, ∇') in L, where σ is a substitution and ∇' is a set of constraints, such that ∇' is satisfiable and for every substitution γ such that $\sigma \circ \gamma$ is ground for Γ, ∇, ∇', the following implication holds: $(\sigma \circ \gamma)$ satisfies $\nabla' \implies (\sigma \circ \gamma)$ is a solution for (Γ, ∇).

A set M of unifiers is *complete*, iff for every solution ρ of (Γ, ∇), there is a unifier $(\sigma, \nabla') \in M$ such that there is a ground substitution γ with $A\sigma\gamma = \rho(A)$, $S\sigma\gamma \sim_\alpha \rho(S)$, $D\sigma\gamma \sim_\alpha \rho(D)$, and $P\sigma\gamma = \rho(P)$ for all variables A, S, D and P occurring in (Γ, ∇) (we say (σ, ∇') *covers* ρ). A unifier (σ, ∇') is a *most general unifier* of (Γ, ∇), iff $\{(\sigma, \nabla')\}$ is a complete set of unifiers for (Γ, ∇).

▶ **Theorem 2.7** ([37, 6, 18, 17]). *The nominal unification problem in NL_{aS}, with ∇ consisting of freshness constraints only, is solvable in quadratic time and is unitary: For a solvable nominal unification problem (Γ, ∇), there exists a most general unifier of the form (σ, ∇') which can be computed in polynomial time.*

3 Preparations for NL_{aASD}-Unification

As a preparation for the unification rules treating equations of the form $D_1[e_1] \doteq D_2[e_2]$, we analyze properties of contexts and expressions in this section. Clearly, for every NL_a-expression e there is some e' with $e \sim_\alpha e'$ s.t. e' satisfies the DVC. If e satisfies the DVC, then $\pi\cdot e$ also satisfies the DVC for any permutation π.

▶ **Lemma 3.1.** *Let e_1, e_2 be two expressions in NL_a that satisfy the DVC (separately). Then $e_1 \sim_\alpha e_2$ is equivalent to the condition that there exists a permutation π with $e_1 = \pi\cdot e_2$, where $supp(\pi) \subseteq (\mathcal{A}t(e_1) \cup \mathcal{A}t(e_2)) \setminus (FA(e_1) \cup FA(e_2))$.*

Proof. If e_1, e_2 satisfy the DVC and $e_1 = \pi\cdot e_2$ where π does not change free atoms of e_1, e_2, then clearly $e_1 \sim_\alpha e_2$. We prove the other direction of the claim by induction on the size. For constants and atoms, this is trivial, since π does not change free atoms. If

$e_1 = \lambda a.e_1'$ and $e_2 = \lambda a.e_2'$, then $e_1' \sim_\alpha e_2'$, hence $e_1' = \pi \cdot e_2'$, for a (minimal) permutation π. Hence $e_1 = \pi \cdot e_2$. Let $e_1 = \lambda a.e_1'$ and $e_2 = \lambda b.e_2'$, with $a \neq b$. Then $a \# e_2$, $a \# e_2'$, and $e_1' \sim_\alpha (a\ b) \cdot e_2'$. The expressions e_1' and $(a\ b) \cdot e_2'$ satisfy the DVC, by induction hypothesis, $e_1' = \pi' \cdot (a\ b) \cdot e_2'$, for a permutation π' where $\pi'(a) = a$. Let π be the permutation $\pi' \cdot (a\ b)$. Then $\pi' \cdot (a\ b) \cdot b = a$. Hence $\pi \cdot e_2 = e_1$. If $e_1 = f\ e_{1,1} \ldots e_{1,n}$, and $e_2 = f\ e_{2,1} \ldots e_{2,n}'$, then by induction there are permutations π_i such that $\pi_i \cdot e_{2,i} = e_{1,i}$ for all i. Since the permutations can be chosen minimal and are only determined by the binders, and since the DVC is assumed, the permutations are disjoint. Thus we can compose (i.e. union) the permutations, and obtain $\pi = \pi_1 \ldots \pi_n$ as the required permutation. ◀

▶ **Remark 3.2.** *The inductive definition of $\sim_\alpha$ for abstractions is*

$$\frac{a \# \lambda b.s_2, s_1 \sim_\alpha (a\ b) \cdot s_2}{\lambda a.s_1 \sim_\alpha \lambda b.s_2}$$

where a may be equal to b or different. Generalizing this for arbitrary contexts results in the situation $C_1 = \lambda a_1 \ldots \lambda a_n.[\cdot]$, $C_2 = \lambda b_1 \ldots \lambda b_n.[\cdot]$, and the rule

$$\frac{CA(C_1) \# C_2[s_2], \exists \pi : (C_1 \sim_\alpha \pi \cdot C_2, s_1 \sim_\alpha \pi \cdot s_2)}{C_1[s_1] \sim_\alpha C_2[s_2]}$$

where $\pi \cdot b_i = a_i$ for all i, $CA(C_1) = \{a_1, \ldots, a_n\}$, and $CA(C_2) = \{b_1, \ldots, b_n\}$. We assume that $a_i \neq a_j$, $b_i \neq b_j$ for $i \neq j$, but $a_i = b_j$ for some i, j may hold. We show below, that the latter rule is already the general one, provided the DVC holds for $C_1[s_1]$ and $C_2[s_2]$.

We now analyze the decomposition of context applications $C[e]$ under the assumption that the DVC holds. For an NL_a-context C, we denote with $CAO(C)$ the ordered tuple of the atoms in $CA(C)$, where the atom ordering is according to the nesting of active bindings: The outermost bound atom comes first. For example, if $C = \lambda a.\lambda a.(f\ (\lambda b.\lambda c.b)\ (\lambda b.\lambda b.[\cdot]))$, then $CAO(C) = (a, b)$.

▶ **Proposition 3.3.** *Let C_1, C_2 be NL_a-contexts and e_1, e_2 be NL_a-expressions, such that C_1 and C_2 have identical hole positions, and $C_1[e_1]$ as well as $C_2[e_2]$ satisfy the DVC. Then $C_1[e_1] \sim_\alpha C_2[e_2]$ is equivalent to*

> *$\forall a \in CA(C_1)$: $a \# C_2[e_2]$ and there is a permutation π with $C_1 \sim_\alpha \pi \cdot C_2$ and $e_1 \sim_\alpha \pi \cdot e_2$, where π does not change free atoms in $C_2[e_2]$, π maps $CAO(C_2)$ to $CAO(C_1)$, and $supp(\pi) \subseteq CA(C_2) \cup CA(C_1)$.*

Proof. We show " $\implies$ " by induction on the length of the hole path of C_1. If the length is 0, then the claim is trivial. For the induction step, let the length be strictly greater than 0.

- If $C_1 = f\ e_1 \ldots \underbrace{C_1'}_{k} \ldots e_n$ then $C_2 = f\ e_1' \ldots \underbrace{C_2'}_{k} \ldots e_n'$. The capture condition holds, since $CA(C_i') = CA(C_i)$. The assumption and the congruence property of $\sim_\alpha$ imply $e_i \sim_\alpha e_i'$ for all $i \neq k$. By the induction hypothesis there is a permutation π_k satisfying the theorem, which is the required permutation.

- If $C_1 = \lambda a.C_1'$ and $C_2 = \lambda a.C_2'$, then the capture condition holds, $C_1'[e_1]$ and $C_2'[e_2]$ are α-equivalent, and we can apply the induction hypothesis.

- If $C_1 = \lambda a.C_1'$ and $C_2 = \lambda b.C_2'$, then $a \# C_2'[e_2]$, and $C_1'[e_1] \sim_\alpha (a\ b) \cdot C_2'[e_2]$. The DVC also holds for $(a\ b) \cdot C_2'$, hence we can apply the induction hypothesis, and obtain $C_1' \sim_\alpha \pi' \cdot (a\ b) \cdot C_2'$, and $e_1 \sim_\alpha \pi' \cdot (a\ b) \cdot e_2$, and $\forall c \in CA(C_1')$: $c \# (a\ b) \cdot C_2'[e_2]$. The equation $c = a$ is not possible, since $C_1[\cdot]$ satisfies the DVC; $c = b$ may be possible, but

since $a\#C_2'[e_2]$, and due to the application of $(a\ b)$ there are no free occurrences of b in $(a\ b)\cdot C_2'[e_2]$. This implies $\forall c \in CA(C_1)$: $c \# C_2[e_2]$. The application $\pi'\cdot(a\ b)\cdot b$ results in a, since π' does not change a. Hence the required permutation is $\pi = \pi'\cdot(a\ b)$.

The direction "$\Longleftarrow$" is easy: if there are two expressions $e_1 \sim_\alpha e_2$, $C_1[e_1], C_2[e_2]$ satisfy the DVC, and there is a permutation π that does not change free atoms in e_2, and $C_1 \sim_\alpha \pi\cdot C_2$, then $C_1[e_1] \sim_\alpha C_2[e_2]$. ◀

▶ **Example 3.4.** The DVC is required in Proposition 3.3: Let $C_1 = f\ a\ \lambda a.[\cdot]$ and $C_2 = f\ a\ \lambda b.[\cdot]$, which implies $C_1[a] = (f\ a\ \lambda a.a) \sim_\alpha (f\ a\ \lambda b.b) = C_2[b]$. The DVC is violated for the left expression. We see that there does not exist a (common) permutation π with $C_1 \sim_\alpha \pi\cdot C_2$ and $e_1 \sim_\alpha \pi\cdot e_2$.

Note that exploiting the general decomposition property of context-variables above in a unification algorithm, even under strong restrictions, requires permutation-variables, and also constraints of the form $CA(D)\#e$. There are investigations on nominal unification permitting variable permutations [7, 10], but an extension to context-variables is open. We will use further components in the constraint set, which will refine the information.

▶ **Definition 3.5.** Let ρ be a ground substitution. The unification algorithm uses the following further constraint components:
- $A \neq e$, where e is an atom expression. (short form of $A\#e$)
- $CA(D)\#e$, which is satisfied by ρ, if for all atoms $a \in CA(D\rho)$: $a\#e\rho$.
- $supp(\pi)\#e$, which is satisfied by ρ, if for all atoms $a \in (supp(\pi)\rho)$: $a\#e\rho$.
- $supp(\pi) \subseteq CA(C_1) \cup CA(C_2)$ which is satisfied by ρ, if for all atoms $a \in (supp(\pi)\rho)$: $a \in CA(C_1\rho) \cup CA(C_2\rho)$.
- $C \neq \emptyset$, which is satisfied by ρ, if $C\rho$ is not the trivial context.

▶ **Proposition 3.6.** *In NL-languages containing expression-variables, freshness constraints can in linear time be encoded as DVC-constraints by translating $a\#e$ into DVC$((f\ (\lambda a.a)\ S))$, plus the equation $S \doteq e$; and $A\#e$ into DVC$((f\ (\lambda A.A)\ S))$ plus the equation $S \doteq e$ where f is a binary function symbol, and in both cases, S is a new expression-variable.*

4   The Unification Algorithm NomUnifyASD for NL_{aASD}

The nominal unification problem in the language NL_{aASDP} without any restriction seems to be too hard (at least we did not find an algorithm to solve it). We discuss the reasons that make the problem so hard, and which restrictions we introduce to handle it. One hint is that already context unification for first-order terms is a quite hard problem. Its solvability was open for decades and recently shown to be in PSPACE [14]. That is why we restrict the input and allow only single occurrences of the same context-variable. A further complication is permutation-variables, since nominal unification with permutation-variables but without contexts, known as equivariant unification [7, 10], is known to be solvable in EXPTIME. However, if context-variables and permutations are combined, then it is unclear how to do constraint solving, since the (to be guessed) support of the permutations depends on the set of captured atoms occurring in the instance of context-variables, and it is unknown how to bound this number of atoms. For this reason, we forbid permutation variables in the input problem (however allow them during execution of the unification algorithm) and consider the specific class of so-called ASD1-unification problems:

▶ **Definition 4.1** (ASD1-Unification Problem)**.** An *ASD1-unification problem* is a nominal unification problem (Γ, ∇) in NL_{aASD} (see Definition 2.6), where each context variable D occurs at most once in Γ and all top-expressions e_i of equations $(e_1 \doteq e_2) \in \Gamma$ have a DVC-constraint DVC$(e_i) \in \nabla$ (for $i = 1, 2$).

We describe a unification algorithm to solve ASD1-unification problems. During the execution of the algorithm the invariant, that context variables occur only once in the equations, must be kept and thus instantiations within the equations (which could duplicate occurrences of context variables) are not permitted. For example, transitions using replacement starting with $S \doteq D[\ldots], S \doteq e_1, S \doteq e_2, \ldots$ would introduce two occurrences of D, since the result is $D[\ldots] \doteq e_1, D[\ldots] \doteq e_2, \ldots$. As a consequence we propose a Martelli-Montanari-style algorithm [21] that avoids instantiations within the unsolved equations.

The state during unification is a tuple $(\Gamma, \nabla, \theta, \Delta)$ using expressions from NL_{aASDP}, where Γ is a set of sets of expressions, so-called multi-equations, ∇ is a set of freshness and DVC-constraints and further constraints of Definition 3.5, θ is a substitution in triangle-form[3], represented as a set of components, and Δ is a set of context-variables that are assumed to be nonempty. We omit the component Δ, if it is not changed by the unification rules.

Multi-equations $M = \{e_1, \ldots, e_n\}$ will sometimes be written as $e_1 \doteq e_2 \doteq \ldots \doteq e_n$, and we write $\pi \cdot M$ to apply permutation π to all expressions in the multi-set, i.e. $\pi \cdot M$ is the multi-equation $\{\pi \cdot e_1, \ldots, \pi \cdot e_n\}$. We will assume that the expressions in Γ are flattened, using iteratively the rule (flatten) in Fig. 1, i.e., in $(f \ e_1 \ldots e_n), \lambda \pi \cdot X'.e$, and in $D[e]$, the expressions e_i, e are of the form $\pi \cdot X$, where X is an atom- or expression-variable.

For permutations, we assume that there is a compression scheme implementing sharing using an SLP [20] where a permutation is a composition (i.e. like a string) of the basic components $(a\ b), (A\ a), (a\ A), (A\ B), P, P^{-1}$, and the expansion of a representation may be exponentially long, and where the required operations on the permutations like composition and inverting can be done in polynomial time. However, to keep the presentation simple, we do not mention this compression and operations in the unification rules.

▶ **Definition 4.2.** The input of the non-deterministic algorithm NOMUNIFYASD is an ASD1-unification problem (Γ, ∇), where Γ is a set of equations and ∇ a set of DVC- and freshness constraints, both over NL_{aASD}. The internal data structure is a tuple $(\Gamma, \nabla, \theta, \Delta)$, over NL_{aASDP}. The algorithm finishes either with Fail, or, if Γ is empty and no failure rule applies, with a tuple $(\nabla', \theta, \Delta')$. The rules of the algorithm NOMUNIFYASD are shown in Figs. 2, 3, 4, 5, 6, which are partitioned into the following rule sets: The rule (flatten) in Fig. 1 is applied until no longer applicable. The variable-replacement and usual decomposition rules are in Fig. 2, the rules for decomposing multi-equations with expressions of the form $D[\ldots]$, and with function symbols or λ as top symbol are in Fig. 3, the decomposition rules for multi-equations of expressions $D[\ldots]$ are in Fig. 4, where the starred rules (DDPRm*) and (DDFrk*) are not used directly; the rules for guessing context-variables as empty or nonempty are in Fig. 5, and the failure rules are in Fig. 6. Rules (fD), (λD) make one (parallel) decomposition step, where rule (fD) first guesses a common first level of the hole positions of all D_i. Rule (DDPrf) guesses that one context is a prefix of the others; rule (DDPRm) guesses a (maximal) common prefix of the contexts D_i (such that it is a proper prefix of all D_i) and rule (DDFrk*) guesses that D_i fork and the first level of the hole positions of D_i.

The priorities of rule application are the sequence as above, i.e. the rules in Fig. 2, 3, 4, 5, 6. Within the rule sets, for rules in Figs. 2, 3, the priority is the sequence as given in the figures. For the rules in Figs. 4 and 5, within the rule sets the priority is the same. The failure rules can be applied at any time.

---

[3] A *substitution in triangle-form* is a shared representation of a substitution, e.g., the substitution in triangle-form $\{x \mapsto (f\ y\ z), y \mapsto a, z \mapsto \lambda b.b\}$ is the substitution $\{x \mapsto f\ a\ (\lambda b.b), y \mapsto a, z \mapsto \lambda b.b\}$, i.e. the substitution itself is idempotent.

(flatten) $\dfrac{\Gamma \cup \{C[e] \doteq M\}}{\Gamma \cup \{C[S] \doteq M\} \cup \{S \doteq e\}}$ where $C \neq [\cdot]$, e is neither $\pi \cdot S_i$ nor $\pi \cdot A_i$ and S is a fresh variable

■ **Figure 1** The flatten-rule.

(Elm1) $\dfrac{(\Gamma \cup \{e \doteq e \doteq M\}, \nabla, \theta)}{(\Gamma \cup \{e \doteq M\}, \nabla, \theta)}$
(Elm2) $\dfrac{(\Gamma \cup \{\{e\}\}, \nabla, \theta)}{(\Gamma, \nabla, \theta)}$

(Slv1) $\dfrac{(\Gamma \cup \{\pi_1 \cdot S \doteq \pi_2 \cdot V \doteq M\}, \nabla, \theta)}{(\Gamma\sigma \cup \{\pi_2 \cdot V \doteq M\sigma\}, \nabla\sigma, \theta \cup \sigma)}$ if $S \neq V$, V is an S- or A-variable or atom, and $\sigma = \{S \mapsto \pi_1^{-1} \cdot \pi_2 \cdot V\}$

(Slv2) $\dfrac{(\Gamma \cup \{\pi_1 \cdot A \doteq \pi_2 \cdot X \doteq M\}, \nabla, \theta)}{(\Gamma\sigma \cup \{\pi_2 \cdot X \doteq M\sigma\}, \nabla \cup \{A = \pi_1^{-1} \cdot \pi_2 \cdot X\}, \theta \cup \sigma)}$ if $X \neq A$ is an atom or atom-variable, and $\sigma = \{A \mapsto \pi_1^{-1} \cdot \pi_2 \cdot X\}$

(Slv3) $\dfrac{(\Gamma \cup \{\pi_1 \cdot X_1 \doteq \pi_2 \cdot X_2 \doteq M\}, \nabla, \theta)}{(\Gamma \cup \{\pi_1 \cdot X_1 \doteq M\}, \nabla \cup \{X_1 = \pi_1^{-1} \cdot \pi_2 \cdot X_2\}, \theta)}$ if X_1, X_2 are atom-variables s.t. $X_1 = X_2$, or X_1, X_2 are atoms

(Slv4) $\dfrac{(\Gamma \cup \{\pi \cdot S \doteq e \doteq M\}, \nabla, \theta)}{(\Gamma \cup \{e \doteq M\}, \nabla, \theta \cup \{S \mapsto \pi^{-1} \cdot e\})}$ if S does not occur in M, e or Γ

(Mrg) $\dfrac{(\Gamma \cup \{\pi_1 \cdot S \doteq M_1\} \cup \{\pi_2 \cdot S \doteq M_2\}, \nabla, \theta)}{(\Gamma \cup \{S \doteq \pi_1^{-1} \cdot M_1 \doteq \pi_2^{-1} \cdot M_2\}, \nabla, \theta)}$

(Slv5) $\dfrac{(\Gamma \cup \{S \doteq \pi \cdot S \doteq M\}, \nabla, \theta)}{(\Gamma \cup \{S \doteq M\}, \nabla \cup \{supp(\pi)\#S\}, \nabla, \theta)}$

(ff) $\dfrac{(\Gamma \cup \{(f\ e_1 \ldots e_{ar(f)}) \doteq (f\ e'_1 \ldots e'_{ar(f)}) \doteq M\}, \nabla, \theta)}{(\Gamma \cup \{(f\ e_1 \ldots e_{ar(f)}) \doteq M\} \cup \{e_1 \doteq e'_1, \ldots, e_{ar(f)} \doteq e'_{ar(f)}\}, \nabla, \theta)}$

(Abstr) $\dfrac{(\Gamma \cup \{\lambda X.e_1 \doteq M\}, \nabla, \theta)}{(\Gamma \cup \{\lambda A'_1.e_1 \doteq M\}, \nabla \cup \{A'_1 = X\}, \theta)}$ if X is of the form a or $\pi \cdot A$ where π is not trivial, and A'_1 is fresh

($\lambda\lambda$) $\dfrac{(\Gamma \cup \{\lambda A_1.e_1 \doteq \lambda A_2.e_2 \doteq M\}, \nabla, \theta)}{(\Gamma \cup \{\lambda A_1.e_1 \doteq M, e_1 \doteq (A_1\ A_2) \cdot e_2\}, \nabla \cup \{A_1 \# \lambda A_2.e_2\}, \theta)}$

■ **Figure 2** Rules of NomUnifyASD for Variables and Decomposition.

(eD) $\dfrac{(\Gamma \cup \{e_1 \doteq (\pi \cdot D)[e_2] \doteq M\}, \nabla, \theta, \Delta)}{(\Gamma \cup \{e_1 \doteq e_2 \doteq M\}, \nabla[[\cdot]/D], \theta \cup \{D \mapsto [\cdot]\}, \Delta)}$ if $tops(e_1)$ is an atom variable, $\texttt{atom}$, or $tops(e_1) = S$, S occurs in e_2; $D \notin \Delta$

(fD) $\dfrac{(\Gamma \cup \{f\ e_1 \ldots e_n \doteq (\pi_1 \cdot D_1)[e'_1] \doteq \ldots \doteq (\pi_m \cdot D_m)[e'_m]\}, \nabla, \theta, \Delta)}{\substack{(\Gamma \cup \{\{e_k\} \cup \{D_{i,1}[e'_i] \mid k = j(i), i \in \{1,..,m\}\} \mid k = 1, \ldots, n\}, \\ \nabla, \theta \cup \{D_i \mapsto \pi_i^{-1} \cdot (f\ e_1 \ldots \underbrace{D_{i,1}}_{j(i)} \ldots e_n) \mid i = 1, \ldots, m\}, \Delta)}}$

if $\forall i : D_i \in \Delta$, and where context variables $D_{i,1}$ for $i = 1, \ldots, m$ are fresh and where for all $i = 1, \ldots, m$, the index position $j(i)$ of D_i is guessed.

(λD) $\dfrac{(\Gamma \cup \{\lambda X.e_0 \doteq (\pi_1 \cdot D_1)[e_1] \doteq \ldots \doteq (\pi_m \cdot D_m)[e_m]\}, \nabla, \theta, \Delta)}{\substack{(\Gamma \cup \{e_0 \doteq (X\ A_1) \cdot (D_{1,1}[e_1]) \doteq \ldots \doteq (X\ A_m) \cdot (D_{m,1}[e_m])\}, \\ \nabla \cup \{X \# \lambda A_i.D_{i,1}[e_i] \mid i = 1, \ldots, m\}, \theta \cup \{D_i \mapsto \pi_i^{-1} \cdot (\lambda A_i.D_{i,1})\}, \Delta)}}$

if $\forall i : D_i \in \Delta$ and X is an atom or atom-variable and $A_i, D_{i,1}$ are fresh

■ **Figure 3** Rules of NomUnifyASD for F-D-Decomposition.

$(\text{DDPrf}) \dfrac{(\Gamma \cup \{(\pi_1 \cdot D_1)[e_1] \doteq (\pi_2 \cdot D_2)[e_2] \doteq \ldots \doteq (\pi_n \cdot D_n)[e_n]\}, \nabla, \theta, \Delta)}{\begin{array}{l}(\Gamma \cup \{e_1 \doteq P_2 \cdot ((\pi_2 \cdot D_{2,2})[e_2]) \doteq \ldots \doteq P_n \cdot ((\pi_n \cdot D_{n,2})[e_n])\}, \\ \nabla \cup \{CA(D_1) \# \pi_1^{-1} \cdot ((\pi_i \cdot D_i)[e_i]), i = 2, \ldots, n\} \\ \quad \cup \{supp(P_i) \subseteq (CAO(\pi_1 \cdot D_1) \cup CAO(\pi_i \cdot D_{i,1})) \mid i = 2, \ldots, n\}, \\ \theta \cup \{D_i \mapsto D_{i,1} D_{i,2}, D_{i,1} \mapsto \pi_i^{-1} \cdot P_i^{-1} \cdot \pi_1 \cdot D_1 \mid i = 2, \ldots, n\}, \\ \Delta \cup \{D_{i,1} \mid i = 2, \ldots, n\}) \text{ where } D_{i,1}, D_{i,2}, P_2, \ldots, P_n \text{ are fresh.}\end{array}}$ if $\forall i{:}D_i \in \Delta$

(DDPRm) First apply (DDPRm*); then apply rule (DDFrk) to the resulting multi-equation.

(DDFrk) Apply (DDFrk*), then remove all introduced variables $S_{i,j}$ using (Slv4)

$(\text{DDPRm*}) \dfrac{(\Gamma \cup \{(\pi_1 \cdot D_1)[e_1] \doteq (\pi_2 \cdot D_2)[e_2] \doteq \ldots \doteq (\pi_n \cdot D_n)[e_n]\}, \nabla, \theta, \Delta)}{\begin{array}{l}(\Gamma \cup \{(\pi_1 \cdot D_{1,1})[e_1] \doteq P_2 \cdot ((\pi_2 \cdot D_{2,1})[e_2]) \doteq \ldots \doteq P_n \cdot ((\pi_n \cdot D_{n,1})[e_n])\}, \\ \nabla \cup \{CA(D_{1,0}) \# (\pi_2 \cdot D_2)[e_2], \ldots, CA(D_{n,0}) \# (\pi_n \cdot D_n)[e_n]\} \\ \quad \cup \{supp(P_i) \subseteq (CAO(D_{1,0}) \cup CAO(D_{i,0})) \mid i = 2, \ldots, n\}, \\ \theta \cup \{D_1 \mapsto (\pi_1^{-1} \cdot D_{1,0}) D_{1,1}, \ldots, D_n \mapsto (\pi_n^{-1} \cdot D_{n,0}) D_{n,1}, \\ \quad D_{2,0} \mapsto P_2^{-1} \cdot D_{1,0}, \ldots, D_{n,0} \mapsto P_n^{-1} \cdot D_{1,0}\}, \\ \Delta \cup \{D_{i,j} \mid i = 1, \ldots, n, j = 0, 1\}) \text{ where } P_i, D_{i,j} \text{ are fresh.}\end{array}}$ if $\forall i{:}D_i \in \Delta$

$(\text{DDFrk*}) \dfrac{(\Gamma \cup \{(\pi_1 \cdot D_1)[e_1] \doteq \ldots \doteq (\pi_n \cdot D_n)[e_n]\}, \nabla, \theta, \Delta)}{\begin{array}{l}(\Gamma \cup \{\{(\pi_i \cdot D_i')[e_i] \mid i \in M_1\} \cup \{\pi_i \cdot S_{i,1} \mid i \notin M_1\}\} \\ \quad \uplus \ldots \uplus \\ \quad \{\{(\pi_i \cdot D_i')[e_i] \mid i \in M_m\} \cup \{\pi_i \cdot S_{i,m} \mid i \notin M_m\}\}, \nabla, \theta \cup \sigma, \Delta)\end{array}}$ if $\forall i{:}\ D_i \in \Delta$

where f with $ar(f) \geq 2$ and the index positions $j(i)$ for $i = 1, \ldots, n$ are guessed such that $|\{j(i) \mid 1 \leq i \leq n\}| \geq 2$; and $M_k := \{h \mid j(h) = k\}$ for $k = 1, \ldots, m$; and $\sigma = \{D_i \mapsto (f\ S_{i,1} \ldots \underbrace{D_i'[\cdot]}_{j(i)} \ldots S_{i,m}) \mid 1 \leq i \leq n\}$ where $D_i', S_{i,i'}$ are fresh.

▪ **Figure 4** Rules of NomUnifyASD for D-D-decomposition.

$(\text{GuessDEmpty}) \dfrac{(\Gamma, \nabla, \theta, \Delta)}{(\Gamma[[\cdot]/D], \nabla[[\cdot]/D], \theta \cup \{D \mapsto [\cdot]\}, \Delta)}$ If $D \notin \Delta$, D occurs in Γ

$(\text{GuessDNonEmpty}) \dfrac{(\Gamma, \nabla, \theta, \Delta)}{(\Gamma, \nabla, \theta, \Delta \cup \{D\})}$ If $D \notin \Delta$, D occurs in Γ

▪ **Figure 5** Rules of NomUnifyASD for guessing D empty or nonempty.

$(\text{Clash}) \dfrac{(\Gamma \cup \{e_1 \doteq e_2 \doteq M\}, \nabla, \theta)}{Fail}$ if $tops(e_1)$ and $tops(e_2)$ are different atoms; or $tops(e_1)$ and $tops(e_2)$ are $\texttt{atom}$, λ or a function symbol, and $tops(e_1) \neq tops(e_2)$; or $tops(e_1)$ is an atom or atom-variable, and $tops(e_2)$ is λ or $f \in \mathcal{F}$.

$(\text{Cycle}) \dfrac{(\Gamma \cup \{S_1 \doteq e_1 \doteq M_1, \ldots, S_n \doteq e_n \doteq M_n\}, \nabla, \theta, \Delta)}{Fail}$

if all e_i are neither variables nor suspensions, all context variables occurring in e_i are in Δ, S_{i+1} occurs in e_i for $i = 1, \ldots, n-1$, and S_1 occurs in e_n

$(\text{eDFail}) \dfrac{(\Gamma \cup \{e_1 \doteq (\pi \cdot D)[e_2] \doteq M\}, \nabla, \theta, \Delta)}{Fail}$ if $tops(e_1)$ is $\texttt{atom}$ or an atom-variable, or $tops(e_1) = S$ and S occurs in e_2; $D \in \Delta$

▪ **Figure 6** Failure Rules of NomUnifyASD.

▶ **Example 4.3.** For $f\,a\,S_2\,S_1 \doteq f\,S_1\,(\lambda a.S_3)\,S_3$ there is a substitution that equates the expressions. However, if there are DVC-constraints for the top expressions then these cannot be satisfied, since for example, the instantiation $\rho(S_3) = a$ cannot be α-renamed. Another example is $f\,S\,S \doteq f\,(\lambda a.a)\,(\lambda b.b)$, which is solvable by $\{S \mapsto \lambda a.a\}$: it does not lead to a DVC-violation, since it is treated as instantiation modulo α.

We define the non-deterministic algorithm that checks satisfiability of the output constraints, where we give the justification later in Section 5. The algorithm exploits the execution sequence of NOMUNIFYASD, since without this information a decision algorithm appears to be impossible since permutation-variables as well as context-variables appear in the constraint.

▶ **Definition 4.4.** The algorithm NOMFRESHASD operates on the output (∇, θ, Δ) of NOMUNIFYASD and uses the set of all atoms and atom-variables occurring in the execution sequence H leading to this output, and the number d of context-variables in the input. It performs the following steps:

(I) Iteratively guess the solution of atom-variables, i.e. for an atom-variable A guess that A is mapped by the solution to an already used atom in H, or to a fresh one, and replace the atom-variable A accordingly in H. In the next iteration the fresh atom is among the used ones. Let H' be the adjusted execution sequence. Note that the exact names of fresh atoms are irrelevant. Thus there is only a linear number (w.r.t. the number of used atoms) of possibilities for every atom-variable. Let M_A be the set of all atoms in the execution sequence H'.

(II) Replace every expression-variable S that occurs in H' and that is not instantiated by θ, by a constant c from the signature.

(III) Construct M_∞ as a set of $|M_A| * (d!)^2$ atoms by extending the set M_A by further fresh atoms, where d is the number of context-variables in Γ. Guess for every context-variable D that occurs in Γ, ∇, θ and that is not instantiated by θ, the ordered set of captured atoms from the given set of atoms.

(IV) Guess the permutation-variables as bijections on the set M_∞.

(V) Test the freshness constraints, equality, disequality, extended freshness constraints, and non-emptiness constraints, which are now immediately computable. To test whether the θ violates the DVC in Γ, use dynamic programming to compute the sets FA, BA for every expression- and context-variable, and $CA(D)$ for the context-variables D that are not instantiated by θ. Then test the DVC-property, which is possible in polynomial time.

5   Properties of NomUnifyASD and NomFreshASD

An example which shows that implicitly requiring the DVC is not stable, in contrast to requiring explicit DVC-constraints, is $\{S_1 \doteq f\,(\lambda a.a)\,(\lambda a.a)\}$. It does not satisfy the DVC, but after applying (flatten), we obtain $\{S_1 \doteq (f\,S\,(\lambda a.a)), S \doteq \lambda a.a\}$ which has the solution $\{S \mapsto \lambda a.a, S_1 \mapsto (f\,(\lambda a'.a')\,(\lambda a.a))\}$. If the initial set ∇ contains DVC($f\,(\lambda a.a)\,(\lambda a.a)$), then there is no solution before and after flattening, since (flatten) cannot be applied to expressions within ∇.

▶ **Lemma 5.1.** *If the input is (Γ, ∇), then the application of (flatten) to a subexpression of e of the equations Γ does not change the set of solutions.*

▶ **Proposition 5.2.** *For a nominal unification problem (Γ, ∇) in NL_{aASD} as input, the non-deterministic algorithm NOMUNIFYASD terminates after a polynomial number of steps.*

Proof. Let `maxArity` be the maximal arity of function symbols in the signature. Let $\mu_1 := \mu_{1,1} + 2 * \texttt{maxArity} * \mu_{1,2}$ where $\mu_{1,1}$ is the number of expressions in Γ and $\mu_{1,2}$ is the number of occurrences of function symbols and λ-s in Γ. Let μ_2 be $\mu_{1,1}$ minus the number of multi-equations. Let μ_3 be the pair of the number of occurrences of context-variables in Γ and the number of context-variables in Γ which are not in Δ. Let μ_4 be the number of occurrences of expressions λX, where X is $\emptyset \cdot a$, $\pi \cdot a$, or $\pi \cdot A$ and π is nontrivial. The following table lists the relation between Γ before and after the application of the rule or subalgorithm, where GD(N)E abbreviates rules (GuessDEmpty) and (GuessDNonEmpty).

rule	μ_1 μ_2 μ_3 μ_4	rule	μ_1 μ_2 μ_3	rule	μ_1 μ_2 μ_3
any rule of Fig. 2 except Abstr	$>$	fD	$>$	DDPrf	$=$ $=$ $>$
Abstr	$=$ $=$ $=$ $>$	λD	$>$	DDPRm	$=$ $>$
eD	$=$ $=$ $>$	GD(N)E	$=$ $=$ $>$	DDFrk	$=$ $>$

It can be verified for (Elm1), (Elm2), (Slv1), (Slv2), (Slv3), (Slv4), (Mrg), (Slv5), (ff), (Abstr), ($\lambda\lambda$), and (eD) by a simple check. It is correct for (fD) and (λD), since $f \, e_1 \ldots e_m$ is removed, which counts 2*`maxArity`. (DDPrf) removes one occurrence of a context-variable. (DDFrk) splits the context-directions, and first introduces expression-variables S_i, which are then removed. Hence the number of expressions in multi-equations is the same, but there are more multi-equations. For (DDPRm), it suffices to check (DDFrk). The measure μ_3 is not increased by any rule, and $\mu_2 \leq \mu_1$, and $\mu_4 \leq \mu_1$, hence the number of executions of rules is polynomial. Since there are multiple sub-steps, we have to argue that a single rule application can be done in polynomial time. The number of steps within the rules is polynomial due to the strict decrease w.r.t. the orderings. ◀

▶ **Lemma 5.3.** *If all top expressions of the initial set of equations Γ are restricted by the* DVC*, then this also holds for all top-expressions in the equations in the sequence of rule executions of the algorithm* NomUnifyASD*.*

▶ **Proposition 5.4.** *Inspecting the details of all rules of* NomUnifyASD *shows soundness: The solutions of the final data structure are also solutions of the input. The following rules of the algorithm* NomUnifyASD *do not lose any solutions, i.e. for every solution ρ of the data structure Q before application, there is a solution ρ' of the output data structure Q', such that $\rho(X) \sim_\alpha \rho'(X)$ for all atom-, expression-, context- and permutation-variables X occurring in Q: Rules from Fig. 2, rules (eD), (λD) from Fig. 3, and the failures rules.*

▶ **Proposition 5.5.** *The algorithm* NomUnifyASD *is complete: If ρ is a solution of the intermediate data structure Q, Γ is not empty and no failure rule applies, then there is a possible rule application, such that there is solution ρ' of the output Q', and $\rho(X) \sim \rho'(X)$ for all atom-, expression-, context- and permutation-variables X occurring in Q.*

We now consider the correctness and complexity of NomFreshASD.

▶ **Lemma 5.6.** *Let H be an execution of* NomUnifyASD *starting with $S_0 := (\Gamma_0, \nabla_0, \theta_0, \Delta_0)$ where $\Gamma_0 = \Gamma$, context variables occur at most once, and θ_0, Δ_0 are trivial or empty. Let the sequence H end with $S_{out} = (\emptyset, \nabla_{out}, \theta_{out}, \Delta_{out})$, and let ρ be a solution of the input as well as of the output S_{out}. Then there is also a solution ρ' that uses only a set of atom VA_∞ with $|VA_\infty| \leq |VA| * ((d!)^2)$, where the visible set VA of atoms $VA = \{a \mid a \text{ occurs in } H\} \cup \{A\rho \mid A \text{ occurs in } H\}$, and where d is the number of context-variables in Γ.*

▶ **Proposition 5.7.** *Let $(\nabla_{out}, \theta_{out}, \Delta_{out})$ be the output of* NomUnifyASD *for input (Γ, ∇). The algorithm* NomFreshASD *decides satisfiability of the output in NEXPTIME in the size*

of the input, where the main components are $|(\Gamma, \nabla)| * ((d!)^2)$, where d is the number of context-variables in Γ. For a fixed upper bound on the number of context-variables, satisfiability can be checked in NP time.

▶ **Remark.** There is a complexity jump between freshness constraints and DVC-constraints in NL_{aSD}, since satisfiability of freshness constraints in NL_{aSD} is in PTIME whereas satisfiability of DVC-constraints in NL_{aSD} is NP-hard.

Combining Propositions 5.5, 5.2, and 5.7 shows:

▶ **Theorem 5.8.** *For Γ, ∇ as input the algorithm* NOMUNIFYASD *terminates and is sound and complete. The computation of some output $(\nabla', \theta', \Delta')$ can be done in NP-time, and the collecting version of the algorithm produces at most exponentially many outputs $(\nabla', \theta', \Delta')$. Decidability of solvability of output constraints, and hence of the input, is in NEXPTIME, and if the number of context-variables is fixed, then in NP time.*

▶ **Theorem 5.9.** *Solvability of ASD1-unification problems is in NEXPTIME.*

6 Specializations, Applications and Examples

We consider nominal unification in NL_{aS} with freshness and DVC-constraints extending the result of [37] (see Theorem 2.7) by allowing DVC-constraints and by restricting NOMUNIFYASD

▶ **Theorem 6.1.** *The nominal unification problem in NL_{aS} where freshness- and DVC-constraints are permitted in ∇ is solvable in polynomial time. Moreover, for solvable (Γ, ∇), there exists a most general unifier of the form (∇', θ) which can be computed in polynomial time, i.e., the problem class is unitary.*

Proof. We assume that the algorithm computes in polynomial time a unifier (∇', θ) consisting of a substitution θ and a constraint set ∇', where in addition we assume that the output substitution θ is represented in triangle-form and that it is of polynomial size (see [33] for the technique). Soundness and completeness of computing only a single execution path follows from Proposition 5.4 since there are no permutation-variables and no context-variables.

For the final satisfiability test, we instantiate the expression-variables in the codomain of θ with a constant from the signature. Note that also $\lambda a.a$ could be used if there is no such constant. For expression-variables S, it is possible to compute $FA(S\theta)$ in polynomial time using dynamic programming. The bound atoms in $FA(S\theta)$ are irrelevant, since these will be renamed by the substitution process which is done modulo α. Then the check for every constraint DVC(e), whether $e\theta$ satisfies the DVC, can be performed in polynomial time.   ◀

The application of NOMUNIFYASD to NL_{AS} also yields at most one most general unifier, however, the complexity to check solvability is increased, since already the solvability of freshness constraints in NL_{AS} is NP-hard [33].

We now consider applications and examples.

▶ **Example 6.2.** As a first example, we consider the equation $\lambda A_1.\lambda A_2.A_1 \doteq \lambda A_1.\lambda A_2.A_2$ together with DVC-constraints for both expressions. The algorithm NOMUNIFYASD finds a potential candidate for a solution, which sets $A_1 \mapsto A_2$. However, constraint checking using NOMFRESHASD fails, since for any instantiation which instantiates A_1 and A_2 with the same atom, the DVC does not hold.

As a second example, consider the input equation $\lambda A_1.\lambda A_2.A_1 \doteq \lambda A_2.\lambda A_1.A_2$ together with DVC-constraints for both expressions. The algorithm NOMUNIFYASD finds a potential

candidate for a solution, which is the identity substitution for A_1 and A_2. Algorithm NomFreshASD shows satisfiability of the DVC-constraints, where a requirement is that A_1 and A_2 are set to different atoms.

As a third simple example, consider the equation $D_1[A] \doteq D_2[B]$ with DVC-constraints for the input and the additional constraint $A \neq B$. We consider the execution that applies rule (DDPrf): This results in the potential solution which sets $A \mapsto P_2 \cdot B$, $D_2 \mapsto D_{2,1}$, $D_{2,1} \mapsto P_2^{-1} \cdot D_1$ and adds the constraints $CA(D_1)\#D_2[B]$, $supp(P_2) \subseteq CAO(D_1) \cup CAO(D_{2,1})$ to ∇. Algorithm NomFreshASD detects satisfiability, for instance, by instantiating $A \mapsto a$, $B \mapsto b$, and then guessing $CAO(D_1) = \{a\}$ and $P_2 = (a\ b)$.

We now consider reductions and transformation rules. Most of them in the application domain of functional programming languages require freshness and/or DVC-constraints to exclude invalid instances of the rules.

▶ **Example 6.3.** The rule app $(\lambda A.S)\ S' \to$ let $A = S'$ in S is a sharing-variant of β-reduction. It needs the constraint DVC(app $(\lambda A.S)\ S'$) if let is recursive, to ensure that for the instances, there are no free occurrence of $A\rho$ in $S'\rho$.

The rule let $A = S$ in $D[A] \to$ let $A = S$ in $D[S]$ copies a single expression represented by the variable S to the target position represented by context-variable D. The constraint DVC(let $A = S$ in $D[A]$) prevents capturing in instances, s.t. $A\rho$ is not captured by $D\rho$. The constraint DVC(let $A = S$ in $D[S]$) prevents that $D\rho$ captures atoms that are free in $S\rho$. Since instances $S\rho$ are α-renamed in (let $A = S$ in $D[S]$)ρ, these constraints suffice.

▶ **Example 6.4.** We describe an exemplary unification problem that occurs in correctness proofs of program transformations. A reduction rule in the let-calculus of [2] is let $A_x=($let $A_y=S_y$ in $S_x)$ in $S_r \to$ let $A_y=S_y$ in let $A_x=S_x$ in S_r with DVC-constraint DVC(let $A_x=($let $A_y=S_y$ in $S_x)$ in S_r) that prevents the occurrence of A_y as free atom in S_r, and thus an unwanted capture in the right hand side. To check whether there is a (nontrivial) overlap of the left hand side of the rule with itself (as a transformation) we form the unification equation let $A_x=($let $A_y=S_y$ in $S_x)$ in $S_r \doteq D[$let $A'_x=($let $A'_y=S'_y$ in $S'_x)$ in $S'_r]$ where the context-variable D is intended as a representation of the reduction strategy[4]. For correct application of the rule, the DVC-constraints DVC(let $A_x=($let $A_y=S_y$ in $S_x)$ in S_r) and DVC($D[$let $A'_x=($let $A'_y=S'_y$ in $S'_x)$ in $S'_r]$) are required. We omit the case that $D \mapsto [\cdot]$ and analyze the instantiation which sets $D \mapsto ($let $A_x=D_1$ in $S_r)$. We obtain the equation let $A_y=S_y$ in $S_x \doteq D_1[$let $A'_x=($let $A'_y=S'_y$ in $S'_x)$ in $S']$. Guessing $D_1 \mapsto [\cdot]$ results in let $A_y=S_y$ in $S_x \doteq$ let $A'_x=($let $A'_y=S'_y$ in $S'_x)$ in S'_r. Guessing $A_y \mapsto A'_x$, we obtain as solution $S_y \doteq ($let $A'_y=S'_y$ in $S'_x)$, $S_x \doteq S'_r$. If we alternatively guess $A_y \neq A'_x$, we obtain as solution $S_y \doteq ($let $A'_y=S'_y$ in $S'_x)$ and $S_x \doteq (A_x\ A'_y) \cdot S'_r$ together with the constraint $A_y \# S'_r$.

7 Conclusion and Further Work

We described and analyzed a nominal unification algorithm for a language with higher-order expressions and variables for atoms, expressions and contexts, where unification problems consist of unification equations, freshness and DVC-constraints. Further work is to extend and adapt the unification and constraint solution method to more constructs of higher-order languages, like a recursive-let, or context-classes.

---

[4] In general, the reduction strategy has to be represented by context classes as in [31].

References

1 Andreas Abel and Brigitte Pientka. Higher-order dynamic pattern unification for dependent types and records. In C.-H. Luke Ong, editor, *Proc. 10th TLCA 2011*, volume 6690 of *Lect. Notes Comput. Sci.*, pages 10–26. Springer, 2011. `doi:10.1007/978-3-642-21691-6_5`.

2 Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *Proc. POPL 1995*, pages 233–246, San Francisco, CA, 1995. ACM Press. `doi:10.1145/199448.199507`.

3 Mauricio Ayala-Rincón, Maribel Fernández, Murdoch Gabbay, and Ana Cristina Rocha Oliveira. Checking overlaps of nominal rewriting rules. *Electr. Notes Theor. Comput. Sci.*, 323:39–56, 2016. `doi:10.1016/j.entcs.2016.06.004`.

4 Mauricio Ayala-Rincón, Maribel Fernández, and Daniele Nantes-Sobrinho. Nominal narrowing. In Delia Kesner and Brigitte Pientka, editors, *Proc. FSCD 2016*, volume 52, pages 11:1–11:17, Dagstuhl, Germany, 2016. Schloss Dagstuhl. `doi:10.4230/LIPIcs.FSCD.2016.11`.

5 Henk P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics*. North-Holland, Amsterdam, New York, 1984.

6 Christophe Calvès and Maribel Fernández. A polynomial nominal unification algorithm. *Theor. Comput. Sci.*, 403(2-3):285–306, 2008. `doi:10.1016/j.tcs.2008.05.012`.

7 James Cheney. The complexity of equivariant unification. In *Proc. ICALP 2004*, volume 3142 of *Lect. Notes Comput. Sci.*, pages 332–344. Springer, 2004. `doi:10.1007/978-3-540-27836-8_30`.

8 James Cheney. *Nominal Logic Programming*. PhD thesis, Cornell University, Ithaca, NY, 2004.

9 James Cheney. Relating higher-order pattern unification and nominal unification. In *Proc. UNIF 2005*, pages 104–119, 2005.

10 James Cheney. Equivariant unification. *J. Autom. Reasoning*, 45(3):267–300, 2010. `doi:10.1007/s10817-009-9164-3`.

11 Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies: A tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

12 Maribel Fernández and Albert Rubio. Nominal completion for rewrite systems with binders. In Artur Czumaj, Kurt Mehlhorn, Andrew M. Pitts, and Roger Wattenhofer, editors, *Proc. ICALP 2012 Part II*, volume 7392 of *Lect. Notes Comput. Sci.*, pages 201–213. Springer, 2012. `doi:10.1007/978-3-642-31585-5_21`.

13 Gérard P. Huet. A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci.*, 1(1):27–57, 1975. `doi:10.1016/0304-3975(75)90011-0`.

14 Artur Jez. Context unification is in PSPACE. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Proc. ICALP 2014 Part II*, volume 8573 of *Lect. Notes Comput. Sci.*, pages 244–255. Springer, 2014. `doi:10.1007/978-3-662-43951-7_21`.

15 Yunus Kutz and Manfred Schmidt-Schauß. Most general unifiers in generalized nominal unification. In *Informal Proceedings of UNIF 2017*, 2017.

16 Matthew R. Lakin. Constraint solving in non-permutative nominal abstract syntax. *Logical Methods in Computer Science*, 7(3), 2011. `doi:10.2168/LMCS-7(3:6)2011`.

17 Jordi Levy and Mateu Villaret. An efficient nominal unification algorithm. In Christopher Lynch, editor, *Proc. RTA 2010*, volume 6 of *LIPIcs*, pages 209–226. Schloss Dagstuhl, 2010. `doi:10.4230/LIPIcs.RTA.2010.209`.

18 Jordi Levy and Mateu Villaret. Nominal unification from a higher-order perspective. *ACM Trans. Comput. Log.*, 13(2):10, 2012. `doi:10.1145/2159531.2159532`.

19   Tomer Libal and Dale Miller. Functions-as-constructors higher-order unification. In Delia
          Kesner and Brigitte Pientka, editors, *Proc. FSCD 2016*, volume 52 of *LIPIcs*, pages 26:1–
          26:17. Schloss Dagstuhl, 2016. `doi:10.4230/LIPIcs.FSCD.2016.26`.

20   Markus Lohrey. Algorithmics on SLP-compressed strings: A survey. *Groups Complexity
          Cryptology*, 4(2):241–299, 2012. `doi:10.1515/gcc-2012-0016`.

21   Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans.
          Program. Lang.*, 4(2):258–282, 1982. `doi:10.1145/357162.357169`.

22   Dale Miller. A logic programming language with lambda-abstraction, function variables,
          and simple unification. *J. Log. Comput.*, 1(4):497–536, 1991. `doi:10.1093/logcom/1.4.
          497`.

23   Andrew K. D. Moran, David Sands, and Magnus Carlsson. Erratic fudgets: A semantic
          theory for an embedded coordination language. In *Proc. Coordination 1999*, volume 1594 of
          Lect. Notes Comput. Sci., pages 85–102. Springer, 1999. `doi:10.1007/3-540-48919-3_8`.

24   Tobias Nipkow. Functional unification of higher-order patterns. In *Proc. LICS 1993*, pages
          64–74. IEEE Computer Society, 1993. `doi:10.1109/LICS.1993.287599`.

25   Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In Richard L. Wexelblat,
          editor, *Proc. PLDI 1988*, pages 199–208. ACM, 1988. `doi:10.1145/53990.54010`.

26   Andrew Pitts. Nominal techniques. *ACM SIGLOG News*, 3(1):57–72, 2016. `doi:10.1145/
          2893582.2893594`.

27   Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge
          University Press, 2013.

28   Zhenyu Qian. Linear unification of higher-order patterns. In *Proc. TAPSOFT 1993*, pages
          391–405. Springer, 1993. `doi:10.1007/3-540-56610-4_78`.

29   David Sabel. Alpha-renaming of higher-order meta-expressions. In Wim Vanhoof and
          Brigitte Pientka, editors, *Proc. PPDP 2017*, pages 151–162. ACM, 2017. `doi:10.1145/
          3131851.3131866`.

30   Manfred Schmidt-Schauß, Temur Kutsia, Jordi Levy, and Mateu Villaret. Nominal unifica-
          tion of higher order expressions with recursive let. In Manuel V. Hermenegildo and Pedro
          López-García, editors, *Proc. LOPSTR 2016*, volume 10184 of *Lect. Notes Comput. Sci.*,
          pages 328–344. Springer, 2016. `doi:10.1007/978-3-319-63139-4_19`.

31   Manfred Schmidt-Schauß and David Sabel. Unification of program expressions with re-
          cursive bindings. In James Cheney and German Vidal, editors, *Proc. PPDP 2016*, pages
          160–173. ACM, 2016. `doi:10.1145/2967973.2968603`.

32   Manfred Schmidt-Schauß and David Sabel. Nominal unification with atom and context
          variables – report version. Frank report 59, Institut für Informatik, Goethe-Universität
          Frankfurt am Main, 2018. URL: `http://nbn-resolving.de/urn/resolver.pl?urn:nbn:
          de:hebis:30:3-452767`.

33   Manfred Schmidt-Schauß, David Sabel, and Yunus Kutz. Nominal unification with atom-
          variables. *J. Symbolic Comput.*, 2018. to appear. `doi:10.1016/j.jsc.2018.04.003`.

34   Manfred Schmidt-Schauß, Marko Schütz, and David Sabel. Safety of Nöcker's strictness ana-
          lysis. *J. Funct. Programming*, 18(04):503–551, 2008. `doi:10.1017/S0956796807006624`.

35   Christian Urban. Nominal techniques in Isabelle/HOL. *J. Autom. Reasoning*, 40(4):327–
          356, 2008. `doi:10.1007/s10817-008-9097-2`.

36   Christian Urban and Cezary Kaliszyk. General bindings and alpha-equivalence in nominal
          Isabelle. *Log. Methods Comput. Sci.*, 8(2), 2012. `doi:10.2168/LMCS-8(2:14)2012`.

37   Christian Urban, Andrew M. Pitts, and Murdoch Gabbay. Nominal unification. In *Proc.
          CSL 2003, EACSL 2003, and KGC 2003*, volume 2803 of *Lect. Notes Comput. Sci.*, pages
          513–527. Springer, 2003. `doi:10.1007/978-3-540-45220-1_41`.

38   Joe B. Wells, Detlef Plump, and Fairouz Kamareddine. Diagrams for meaning preservation. In Robert Nieuwenhuis, editor, *Proc. RTA 2003*, volume 2706 of *Lect. Notes Comput. Sci.*, pages 88–106. Springer, 2003. `doi:10.1007/3-540-44881-0_8`.

A   Detailed Proofs

A.1   Completeness of NomUnifyASD

▶ **Definition A.1.** We introduce n-contexts for some $n \geq 1$ (also called multi-contexts) in NL_a. These are expressions of NL_a extended by constants (holes) $[\cdot]_i, i = 1, \ldots, n$, where every hole occurs at most once. For $n \geq 1$ and two n-contexts C_1, C_2 the relation $C_1 \sim_\alpha C_2$ holds, iff for all n-tuples $a_1, \ldots, a_n$ of (perhaps equal) atoms a_i, it holds that $C_1[a_1, \ldots, a_n] \sim_\alpha C_2[a_1, \ldots, a_n]$ as expressions, which is a generalization from contexts to multi-contexts.

The following lemma helps in the completeness proof of rule (DDFrk).

▶ **Lemma A.2.** *Let* $n \geq 1$, C_1, C_2 *be* NL_a-n-*contexts, and* $e_{1,i}, e_{2,i}, i = 1, \ldots, n$ *be* NL_a-*expressions, such that the corresponding hole positions of* C_1 *and* C_2 *are identical, and such that* $C_1[e_{1,1}, \ldots, e_{1,n}]$ *and* $C_2[e_{2,1} \ldots, e_{2,n}]$ *satisfy the* DVC. *Then* $C_1[e_{1,1}, \ldots, e_{1,n}] \sim_\alpha C_2[e_{2,1}, \ldots, e_{2,n}]$ *iff the following holds:*

$CA(C_1) \,\#\, C_2[e_{2,1} \ldots, e_{2,n}]$, and there is a permutation π that does not change free atoms in $C_2[e_{2,1} \ldots, e_{2,n}]$ with $C_1 \sim_\alpha \pi{\cdot}C_2$ and $e_{1,i} \sim_\alpha \pi{\cdot}e_{2,i}$ for all i, π maps $CAO(C_2)$ to $CAO(C_1)$, and $supp(\pi) \subseteq CAO(C_1) \cup CAO(C_2)$.

▶ **Proposition 5.5.** *The algorithm* NOMUNIFYASD *is complete: If* ρ *is a solution of the intermediate data structure* Q, Γ *is not empty and no failure rule applies, then there is a possible rule application, such that there is solution* ρ' *of the resulting data structure* Q', *and* $\rho(X) \sim \rho'(X)$ *for all atom-, expression-, context- and permutation-variables* X *occurring in* Q.

Proof. Let ρ be a solution of the current state. We scan the cases:

1. We can assume that all multi-equations have at least two expressions since otherwise rule (Elm1) is applicable.

2. We can also assume that all context-variables that occur in Γ are contained in Δ, by applying either (eD), or one of the rules (GuessDEmpty) or (GuessDNonEmpty), where the choice is directed by the solution.

3. If there is a multi-equation that has only context-variables as top symbols, then one of the rules from Fig. 4 is applicable, depending on the solution ρ, and there is a solution ρ' of the output that extends ρ. The condition that top-expressions in the input satsify the DVC and that the input are ASD1-unification problems is necessary for the application of Proposition 3.3.

4. If there is a multi-equation such that all but one expression have context-variables as top symbols, then there are several possibilities: Since $D \in \Delta$ for all D, one of the rules from Fig. 3 is applicable, since either the context-variables' instances have a common prefix or not. Proposition 3.3 and the knowledge on permutations show that the execution of the rules is possible. In any case, there will be a solution ρ' after the application that is an extension of ρ (on the variables of Q).

5. For the other cases there are at least two expressions in the multi-equation, which do not have a context-variable as top-symbol. The failure rules are not applicable, since otherwise, there is no solution. If the top symbols of two expressions are λ, or function

symbols, then rules (ff), (Abstr) or ($\lambda\lambda$) can be applied and there is a solution ρ' extending ρ after the application. If there are two expressions of the form $\pi{\cdot}X$, where X is an atom or expression-variable, then (Slv1), (Slv2), (Slv3), or (Slv4) is applicable, and there is a solution ρ' extending ρ after the application. Similar for the case where $tops(.)$ yields $\mathtt{atom}$ twice. If one expression is of the form $\pi{\cdot}X$, and the other is not of this form, then we apply (Slv4) if this is possible.

6. The final case is that for all equations, the equation pattern permits only applications of rule (Slv4), all multi-equations have only two expressions, but the conditions on non-containment of S in rule (Slv4) prevent this. Defining a quasi-ordering on the expression-variables generated by the containment ordering over equations shows that there are no finite instance-expressions for some expression-variable, which is impossible, since we have assumed that there are solutions. Indeed in this case a failure rule would apply. ◀

A.2 Satisfiability of Constraints of NomUnifyASD

▶ **Lemma 5.6.** *Let H be a sequence of executions of* NomUnifyASD *starting with $S_0 :=$ $(\Gamma_0, \nabla_0, \theta_0, \Delta_0)$ where $\Gamma_0 = \Gamma$, the condition on the input are as stated in Definition 4.4, and θ_0, Δ_0 are trivial or empty. Let the sequence H end with $S_{out} = (\emptyset, \nabla_{out}, \theta_{out}, \Delta_{out})$, and let ρ be a solution of the input as well as of the output S_{out}. Then there is also a solution ρ' that uses only a set of atom VA_∞ with $|VA_\infty| \le |VA| * ((d!)^2)$, where the visible set VA of atoms $VA = \{a \mid a \text{ occurs in } H\} \cup \{A\rho \mid A \text{ occurs in } H\}$, and where d is the number of context-variables in Γ.*

Proof. We show a stronger claim by induction on the steps of the execution: Let $\mathcal{D}_F$ and $\mathcal{D}_P$ be the set of context-variables D_1 in the applications of (DDPrf) in H and $D_{1,0}$ in the application of (DDPRm) in H, and P_i, $i = 2, \ldots, n$ in the applications of (DDPrf) and (DDPRm), resp. Let us call these the *focused* context-variables and the *focused* permutation-variables in the respective rule applications. These are the set of context-variables that are moved to the codomain of θ. The permutation-variables are exactly all the generated ones in H. Let $VA = \{a \mid a \text{ occurs in } H\} \cup \{A\rho \mid A \text{ occurs in } H\}$. Then the size of VA is polynomial, since the execution sequence H can be generated in polynomial time according to Proposition 5.2. The claim is: there is a solution ρ' that uses only the set VA_∞ of atoms, where $VA_0 = VA$, and VA_i is constructed below, such that $|VA_{i+1}| \le |VA_i| * d^2$, where d is the number of context-variables in the input, and where $|VA_{i+1}| > |VA_i|$ only if rule (DDPrf) or (DDPRm) was executed. The construction implies $VA_i \subseteq VA_{i+1}$. The final VA_∞ is defined as the final VA_i.

We define the construction: Let i be an index in H and $S_i = (\Gamma_i, \nabla_i, \theta_i, \Delta_i)$ be a state in H with set VA_i, such that the next step is (DDPrf) or (DDPRm) leading to S_{i+1}. Let us assume that it is the first occasion in H such that a focussed context-variable or permutation-variable that is in the focus of a rule application (DDPrf) or (DDPRm), uses an atom a' in its instances under ρ, where $a' \notin VA_i$. Then the following changes are made to the solution ρ, resulting in ρ' and a modified execution sequence H'.

- First consider the modification concerning the rule (DDPrf):
  - Using the same notation as in the rule application, we consider the instances $(CAO(\pi_1{\cdot}D_1)\rho)$, $(CAO(P_2{\cdot}\pi_2{\cdot}D_{2,1})\rho), \ldots, (CAO(P_n{\cdot}\pi_n{\cdot}D_{n,1})\rho)$. We can assume that $(\pi_1{\cdot}D_1)\rho$, $(P_2{\cdot}\pi_2{\cdot}D_{2,1})\rho, \ldots, (P_n{\cdot}\pi_n{\cdot}D_{n,1})\rho$ only consist of $\lambda a_{k,1}, \ldots, a_{k,m}.[\cdot]$ where $m = |CAO(\pi_1{\cdot}D_1)|$, by modifying the solution ρ, which is without effect on the further execution of the algorithm NomUnifyASD and solvability. We show that the number

of binders can be bounded: Luckily, we can also eliminate the binder at position j, if $a_{k,j}$ is not in VA_i for all k: eliminate the binder j in every context above, then modify the instantiation of the permutation-variables P_k such that these map exactly $(\pi_i \cdot D_{i,1})\rho$ to $CAO(\pi_1 \cdot D_1)$ for $k \geq 2$, which is justified by Proposition 3.3. Hence P_k does not need any extra fresh atoms in its support. Using the thus modified ρ, we replace all atoms (as expressions) by the constant c at the following positions: If the atom at this position in the instance $e_i\rho$ is an atom $a_{k,j}$ for some k.

- Since binders cannot be repeated, an upper bound on the maximal number of binders for a single context-variable is $|V_i| * d'$, where d' is the number of context-variables in Γ_i. The number of all used atoms in the instances is at most $|V_i| * d' * d'$.

- Let ρ' be ρ after these modifications. The ground substitution ρ' is a solution of the state S_{i+1}, and such that the same execution still leads to a final state that covers ρ'. There is no effect on the execution of the rules of the algorithm, since the changes are only in the solution.

- Now consider the modification for the rule (DDPRm). It is similar to the previous case, but we detail it, since the names of variables are different.

  - Consider the instances $(CAO(\pi_1 \cdot D_{1,0})\rho), \ldots, (CAO(\pi_n \cdot D_{n,0})\rho)$. We can assume that $(\pi_k \cdot D_{k,0})\rho$ only consist of $\lambda a_{k,1}, \ldots, a_{k,m}.[\cdot]$ where $m = |CAO(D_{1,0})|$, by modifying the solution ρ, which is without effect on the execution of the algorithm NomUnifyASD and solvability. We can also eliminate the binder at position j, if $a_{k,j}$ is not in VA_i for all k, as follows: eliminate the binder j in every $(\pi_k \cdot D_{k,0})\rho$, then modify the instantiation of the permutation-variables P_k such that these map exactly $CAO(D_{k,0})$ to $CAO(D_{1,0})$ for $k \geq 2$. Using the thus modified ρ, we replace atoms by the constant c at all the following positions: If the atom at this position in the instance $e_i\rho$ is an atom with erased binder: $a_{k,j}$ for some k.

  - An upper bound on the maximal number of binders for a single context-variable is $|V_i|*d'*d'$ where d' is the number of context-variables in Γ_i.

  - Let ρ' be ρ after these modifications. The ground substitution ρ' is a solution of the state S_{i+1}, and such that the same execution still leads to a final state that covers ρ'.

The number of executions of rules (DDPrf), (DDPRm) is at most the number of different context-variables. This holds, since (DDPrf) removes one context-variable, and since (DDPRm) calls (DDFrk), and (DDFrk) can be applied also at most as often as there are expressions with topmost context-variables. As additional argument, all other rules keep the number of context-variables, and there is never a merge of two multisets that only contain context-variables. The estimation for the maximal number of variables is that in $(CAO(\pi_1 \cdot D_1)\rho)$, there can at most be $d_i * |V_i|$ variables, where d_i is the number of context-variables in Γ_i. Since there is an iterated multiplication, we obtain $|VA| * d * d * (d-1) * (d-1) \ldots$, which leads to the estimation as claimed.

This change can be iterated until there are no (DDPrf), (DDPRm)-steps having an index j where completely fresh variables are used for all context-variables in the multi-equation. Finally, we have constructed VA_∞, and the modified solution ρ'. ◀

Cumulative Inductive Types In Coq

Amin Timany
imec-Distrinet, KU Leuven, Leuven, Belgium
amin.timany@cs.kuleuven.be

Matthieu Sozeau
Inria Paris & IRIF, Paris, France
matthieu.sozeau@inria.fr

──────── **Abstract** ────────

In order to avoid well-known paradoxes associated with self-referential definitions, higher-order dependent type theories stratify the theory using a countably infinite hierarchy of universes (also known as sorts), $\mathtt{Type}_0 : \mathtt{Type}_1 : \cdots$. Such type systems are called cumulative if for any type A we have that $A : \mathtt{Type}_i$ implies $A : \mathtt{Type}_{i+1}$. The Predicative Calculus of Inductive Constructions (PCIC) which forms the basis of the Coq proof assistant, is one such system. In this paper we present the Predicative Calculus of Cumulative Inductive Constructions (PCuIC) which extends the cumulativity relation to inductive types. We discuss cumulative inductive types as present in Coq 8.7 and their application to formalization and definitional translations.

1 Introduction

In higher-order dependent type theories every type is a term and hence has a type. As expected, having a type of all types which is a term of its own type, leads to inconsistencies such as Girard's paradox [7] and Hurken's paradox [9]. To avoid this, a predicative hierarchy of universes is usually employed. The predicative Calculus of Inductive Constructions (PCIC) at the basis of the Coq proof assistant [16], additionaly supports *cumulativity*: as a Pure Type System with subtyping, it includes the rule: $\Pi\Gamma.\mathtt{Type}_i \leq \Pi\Gamma.\mathtt{Type}_{i+1}$.

Earlier work [15] on universe-polymorphism in Coq allows constructions to be polymorphic in universe levels. The quintessential universe-polymorphic construction is that of categories:

```
Record Categoryi,j := { Obj : Type\[@{i}; Hom : Obj → Obj → Type\]@{j}; ··· }.¹
```

However, PCIC does not extend the subtyping relation (induced by cumulativity) to inductive types. As a result, there is no subtyping relation between distinct instances of a universe-polymorphic inductive type. That is, for a category C, having both C : Category_{i,j} and C : Category_{i',j'} is only possible if i = i' and j = j'.

In this work, we build upon the preliminary and in-progress work of Timany and Jacobs [17] on extending PCIC to PCuIC (predicative Calculus of Cumulative Inductive Constructions).

---

¹ Records in Coq are syntactic a special form of inductive types. Type@{i} is Coq's syntax for $\mathtt{Type}_i$.

In pCuIC, subtyping of inductive types no longer imposes the strong requirement that both instances of the inductive type need to have the same universe levels. In addition, in pCuIC we consider two inductive types, that are in mutual cumulativity relation, to be judgementally equal. This cumulativity relation is also extended to the constructors of inductive types, resulting in a very lax criteria for conversion of constructors. In pCuIC, in order for a term $C : \mathtt{Category_{i,j}}$ to have the type $\mathtt{Category_{i',j'}}$, *i.e.*, for the cumulativity relation $\mathtt{Category_{i,j}} \preceq \mathtt{Category_{i',j'}}$ to hold, it is only required that $\mathtt{i} \leq \mathtt{i'}$ and $\mathtt{j} \leq \mathtt{j'}$. This is indeed what a mathematician would expect when universe levels of the type $\mathtt{Category}$ are thought of as representing (relative) smallness and largeness. For more details on representing relative size reasoning in category theory using universe levels see Timany and Jacobs [18].

Contributions.   Timany and Jacobs [17] give an account of then work-in-progress on extending pCIC with a single rule for cumulativity of inductive types. The authors show the soundness of a rather restricted subsystem their system. In this paper, we extend and complete this work, through the following contributions:

- We extend Timany and Jacobs [17] to support lowering levels as well as lifting them. For instance, given universe levels $i < j$ and a type $A : \mathtt{Type}_i$, the old system of Timany and Jacobs [17] only allowed the subtyping $list_i\ A \preceq list_j\ A$. Our generalization of the subtyping relation for inductive types also allows $list_j\ A \preceq list_i\ A$ and furthermore judgementally equates them.Similarly for constuctors, it justifies $nil_i\ A \simeq nil_j\ A$, rendering universe annotations computationally irrelevant in this case.

- This generalization allows universe polymorphism to subsume the functionality of *template polymorphism*, a feature of Coq which allows under certain conditions two instances of a *non-universe-polymorphic* inductive type at different universe levels to be unified.

- We prove soundness of cumulativity by giving a model in ZFC which builds on the one of Lee and Werner [10]. This model naturally supports cumulativity for inductive types, as most set-theoretic models will. However, the argument for consistency in [10] assumes strong normalization to model recursive functions, which already implies consistency. We solve this problem by resorting to eliminators instead of the fixpoint and case constructs.

- Cumulativity of inductive types as presented in this paper is integrated in the stable version 8.7 of Coq [16]. We discuss remaining issues regarding the replacement of template polymorphism by universe polymorphism with cumulative inductive types.

- We highlight two applications of Cumulative Inductive Types: one to the formalization of the Yoneda lemma, and the other one to the construction of definitional translations / syntactic models of type theories.

Structure of the paper.   In §2 we present the system pCIC. Section 3 discusses universes in pCIC, universe-polymorphic constructions and also how template polymorphism treats monomorphic constructions. In §4 we define the pCuIC and describe how the cumulativity relation is extended to inductive types. In §5 we present our model of pCuIC in ZFC set theory and prove soundness of pCuIC. Section 6 briefly describes the implementation of pCuIC in Coq and §7 two applications of Cumulative Inductive Types. In Section 8 we give a short discussion of related and future work. We conclude with a discussion in §9.

2   Predicative calculus of inductive constructions (pCIC)

In this section we give a short account of the system pCIC, presented with an equality judgment. Note that this system does not feature universe polymorphism. We will discuss universe polymorphism in Section 3. The full system pCuIC (and pCIC being its subsystem) can be found in Timany and Sozeau [19]. The sorts of pCIC are as follows:

$$\frac{\text{WF-\textsc{ctx-hyp}}}{\Gamma \vdash A : s \qquad x \notin \mathsf{dom}(\Gamma)}{\mathcal{WF}(\Gamma, x : A)} \qquad \frac{\text{WF-\textsc{ctx-def}}}{\Gamma \vdash t : A \qquad x \notin \mathsf{dom}(\Gamma)}{\mathcal{WF}(\Gamma, (x := t : A))} \qquad \frac{\textsc{Prop}}{\mathcal{WF}(\Gamma)}{\Gamma \vdash \texttt{Prop} : \texttt{Type}_i}$$

$$\frac{\textsc{Hierarchy}}{\mathcal{WF}(\Gamma) \qquad i < j}{\Gamma \vdash \texttt{Type}_i : \texttt{Type}_j} \qquad \frac{\textsc{Let}}{\Gamma, (x := t : A) \vdash u : B}{\Gamma \vdash \textbf{let } x := t : A \textbf{ in } u : B\,[t/x]} \qquad \frac{\textsc{App}}{\Gamma \vdash M : \boldsymbol{\Pi}x : A.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B\,[N/x]}$$

$$\frac{\textsc{Var}}{\mathcal{WF}(\Gamma) \qquad x : A \in \Gamma \text{ or } (x := t : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\textsc{App-eq}}{\Gamma \vdash M \simeq M' : \boldsymbol{\Pi}x : A.\, B \qquad \Gamma \vdash N \simeq N' : A}{\Gamma \vdash M\ N \simeq M'\ N' : B\,[N/x]}$$

$$\frac{\textsc{Prod}}{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2 \qquad \mathcal{R}_s(s_1, s_2, s_3)}{\Gamma \vdash \boldsymbol{\Pi}x : A.\, B : s_3} \qquad \frac{\textsc{Lam}}{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \boldsymbol{\Pi}x : A.\, B : s}{\Gamma \vdash \boldsymbol{\lambda}x : A.\, M : \boldsymbol{\Pi}x : A.\, B}$$

$$\frac{\textsc{Prod-eq}}{\Gamma \vdash A \simeq A' : s_1 \qquad \Gamma, x : A \vdash B \simeq B' : s_2 \qquad \mathcal{R}_s(s_1, s_2, s_3)}{\Gamma \vdash \boldsymbol{\Pi}x : A.\, B \simeq \boldsymbol{\Pi}x : A'.\, B' : s_3}$$

■ **Figure 1** An excerpt of the typing rules for the basic constructions.

$\texttt{Prop}, \texttt{Set} = \texttt{Type}_0, \texttt{Type}_1, \texttt{Type}_2, \dots$ We write the dependent product (function) type as $\boldsymbol{\Pi}x : A.\, B$. This is the type of functions that given $t : A$, produce a result of type $B\,[t/x]$. We write lambda abstraction in the Church style, $\boldsymbol{\lambda}x : A.\, t$. The term $\textbf{let } x := t : A \textbf{ in } u$ is the Church style let binding. We write function applications as juxtapositions, e.g., $M\ N$. Figure 1 shows an excerpt of the typing rules for these basic constructions.

There are three different judgements in this figure: well formedness of typing contexts $\mathcal{WF}(\Gamma)$, the typing judgement, $\Gamma \vdash t : A$, *i.e.*, term t has type A under the typing context Γ, and judgemental equality, $\Gamma \vdash t \simeq t' : A$, *i.e.*, terms t and t' are judgementally equal terms of type A under the typing context Γ. Most of the basic constructions (wherever it makes sense) come with a rule for judgemental equality. These rules indicate which parts of the constructions are sub-terms that can be replaced by some other judgementally equal term. For example, the rule $\textsc{Prod-eq}$ states that the domain and codomain of (dependent) function types can be replaced by judgementally equal terms. The relation $\mathcal{R}_s(s_1, s_2, s_3)$ determines the sort of the product type based on the sort of the domain and codomain. The relation is defined as follows: $\mathcal{R}_s(\texttt{Type}_i, \texttt{Type}_j, \texttt{Type}_{\max\{i,j\}})$, $\mathcal{R}_s(\texttt{Prop}, \texttt{Type}_i, \texttt{Type}_i)$ and $\mathcal{R}_s(s, \texttt{Prop}, \texttt{Prop})$. Note that the impredicativity of the sort $\texttt{Prop}$ is enforced by this relation.

Inductive types. In this paper we consider blocks of mutual inductive types that live in predicative universes. We avoid inductive types in $\texttt{Prop}$ because they add extra complexity to the construction of set theoretic models. On the other hand, they can be encoded using their Church encoding. For instance, the type $\texttt{False}$ and conjunction of two predicates can be defined as follows:

```
Definition conj (P Q : Prop) := forall (R : Prop), (P → Q → R) → R.
Definition False := forall (P : Prop), P.
```

We write $\mathbf{Ind}_n\,\{\Delta_I := \Delta_C\}$ for an inductive block where n is the number of parameters, Δ_I is list of inductive types of the block and Δ_C is the list of constructors. The arguments of an inductive type that are not parameters are known as *indices*. The following are some

Ind-WF
$$\frac{\mathcal{I}_n(\Gamma, \Delta_I, \Delta_C) \qquad (A \equiv \mathbf{\Pi}p : P. \mathbf{\Pi}m : M. A_d \qquad \Gamma \vdash A : s_d \text{ for all } (d : A) \in \Delta_I)}{(T \equiv \mathbf{\Pi}p : P. T' \qquad \Gamma, \Delta_I, p : P \vdash T' : A_d \text{ for all } (c : T) \in \Delta_C \text{ if } c \in \mathsf{Constrs}(\Delta_C, d))}$$
$$\mathcal{WF}(\Gamma, \mathsf{Ind}_n \{\Delta_I := \Delta_C\})$$

Assuming $\mathcal{D} \equiv \mathsf{Ind}_n \{\Delta_I := \Delta_C\} \in \Gamma$ and $\mathcal{WF}(\Gamma)$:

Ind-type
$$\frac{d_i \in \mathsf{dom}(\Delta_I)}{\Gamma \vdash \mathcal{D}.d_i : \Delta_I(d_i)}$$

Ind-constr
$$\frac{c \in \mathsf{dom}(\Delta_C)}{\Gamma \vdash \mathcal{D}.c : \Delta_C(c) \left[\overrightarrow{\Delta_I.d/\vec{d}}\right]}$$

Ind-Elim
$$\frac{\mathsf{dom}(\Delta_I) = \{d_1, \ldots, d_l\} \qquad \mathsf{dom}(\Delta_C) = \{c_1, \ldots, c_{l'}\}}{\Gamma \vdash Q_{d_i} : \mathbf{\Pi}\vec{x} : \vec{A}.\,(d_i\ \vec{x}) \to s' \text{ where } \Delta_I(d_i) \equiv \mathbf{\Pi}\vec{x} : \vec{A}.\,s \text{ for all } 1 \le i \le l}$$
$$\frac{\Gamma \vdash t : \mathcal{D}.d_k\ \vec{m} \qquad \Gamma \vdash f_{c_i} : \xi_{\mathcal{D}}^{\vec{Q}}(c_i, \Delta_C(c_i)) \text{ for all } 1 \le i \le l'}{\Gamma \vdash \mathsf{Elim}(t; \mathcal{D}.d_k; Q_{d_1}, \ldots, Q_{d_l})\left\{f_{c_1}, \ldots, f_{c_{l'}}\right\} : Q_{d_k}\ \vec{m}\ t}$$

Figure 2 Typing rules for inductive types and eliminators.

of the examples of inductive types written in this format: natural numbers, lists, vectors and a mutually inductive encoding of forests respectively.

$\mathsf{Ind}_0\{nat : \mathsf{Set} := Z : nat, S : nat \to nat\}$

$\mathsf{Ind}_1\{list : \mathbf{\Pi}A : \mathsf{Set}. \mathsf{Set} := nil : \mathbf{\Pi}A : \mathsf{Set}.\, list\ A, cons : \mathbf{\Pi}A : \mathsf{Set}.\, A \to list\ A \to list\ A\}$

$\mathsf{Ind}_1\{vec : \mathbf{\Pi}A : \mathsf{Set}.\, nat \to \mathsf{Set} :=$

$vnil : \mathbf{\Pi}A : \mathsf{Set}.\, vec\ A\ Z, vcons : \mathbf{\Pi}A : \mathsf{Set}.\, \mathbf{\Pi}n : nat.\, A \to vec\ A\ n \to vec\ A\ (S\ n)\}$

$\mathsf{Ind}_0\{FTree : \mathsf{Type}_0, Forest : \mathsf{Type}_0 :=$

$leaf : FTree, node : Forest \to FTree, Fnil : Forest, Fcons : FTree \to Forest \to Forest\}$

Figure 2 shows the typing rules for inductive types and their eliminators. Rule `Ind-WF` describes when an inductive type is well-formed. Here, A_{d_i} is a sort that is called the arity of the inductive type d_i. This rule requires that all inductive types and constructors of the block are well-typed. The set $\mathsf{Constrs}(\Delta_C, d)$ is the set of constructors in Δ_C that produce something of type d. The proposition $\mathcal{I}_n(\Gamma, \Delta_I, \Delta_C)$ describes the syntactic constraints for well-formedness of an inductive block. For precise details see our extended technical appendix [19]. It requires that all inductive types and all constructors of the block have as their first arguments the parameters of the block, e.g., A in *list* above. The parameters must be fixed for the whole block. In particular, the codomain type of each constructor must construct an inductive type that is applied to the parameters of the block, i.e., every constructor of *list* must construct a term of type *list A*. All inductive types above satisfy these criteria. Both constructors of the type *vec*, for instance, start with the argument $A : \mathsf{Type}_0$ and also they both construct a vector *vec A n* for some natural number n. Moreover, all arguments of constructors that are vectors take the same parameter A. This is the essential difference between parameters and indices. In addition, $\mathcal{I}_n(\Gamma, \Delta_I, \Delta_C)$ also requires that all occurrences of inductive types of the block in any of the constructors of the block are strictly positive.

▶ Remark. Note that the names of inductive types and constructors of an inductive block in a typing context are *not* part of the domain of that context. We never refer to an inductive type or constructor without mentioning the block. In particular, we require for well-formed contexts that no variable appears in the domain of the context more than once. This restriction does not apply to inductive types and constructors in mutual inductive blocks.

$$\frac{\text{BETA}}{\Gamma, x : A \vdash M : B \qquad \Gamma, x : A \vdash B : s \qquad \Gamma \vdash N : A}{\Gamma \vdash (\boldsymbol{\lambda} x : A.\, M)\, N \simeq M\,[N/x] : B\,[N/x]}$$

$$\frac{\text{ETA}}{\Gamma \vdash t : \boldsymbol{\Pi} x : A.\, B}{\Gamma \vdash t \simeq \boldsymbol{\lambda} x : A.\, t\, x : \boldsymbol{\Pi} x : A.\, B}$$

■ **Figure 3** An excerpt of judgemental equality rules.

Eliminators. In this work, we consider eliminators for inductive types as opposed to CoQ's structurally recursive definitions, *i.e.*, `Fixpoint`s and `match` blocks in CoQ. Note, however, that these can be encoded using eliminators as they are presented here [12] using the accessibility proof of the subterm relation, definable for any (non-propositional) inductive family.

Rule IND-ELIM in Figure 2 describes the typing for eliminators. Inductive types in a mutual inductive block can appear in one another. Hence, we define the elimination of inductive types for the entire block. We write $\textbf{Elim}(t; \mathcal{D}.d_k; Q_{d_1}, \ldots, Q_{d_l})\left\{ f_{c_1}, \ldots, f_{c_{l'}} \right\}$ for the elimination of t that is of type of the inductive type $\mathcal{D}.d_k$ (applied to values for parameters and indices). The term Q_{d_i} is the *motive* of elimination for the inductive type $\mathcal{D}.d_i$. This is basically a function that given the $\vec{a}$ and u such that u has type $\mathcal{D}.d_i\ \vec{a}$ produces a type (a term of some sort s'). The idea is that eliminating the term u should produce a term of type $Q_{d_i}\ \vec{a}\ u$. Note that the elimination $\textbf{Elim}(t; \mathcal{D}.d_k; Q_{d_1}, \ldots, Q_{d_l})\left\{ f_{c_1}, \ldots, f_{c_{l'}} \right\}$ is a term of type $Q_{d_k}\ \vec{b}\ t$ where t has type $d_k\ \vec{b}$.

In the elimination above the terms f_{c_i} are *case-eliminators*. The case-eliminator f_{c_i} is a function that describes the elimination of terms that are constructed using the constructor c_i. The term f_{c_i} is a function. It takes arguments of the constructor c_i together with the result of elimination of the (mutually) recursive arguments and produces a term of the appropriate type (according to the motives). This type is exactly what is formally defined as the type of the case eliminator for constructor c_i, $\xi_{\mathcal{D}}^{\vec{Q}}(c_i, \Delta_C(c_i))$. The formal definition of the types of case-eliminators can be found in Timany and Sozeau [19]. A simple example of eliminator is the induction principle for natural numbers:

$$\boldsymbol{\lambda} P : nat \to \texttt{Prop}.\, \boldsymbol{\lambda} pz : P\, Z.\, \boldsymbol{\lambda} ps : \boldsymbol{\Pi} x : nat.\, P\, x \to P\, (S\, x).\, \boldsymbol{\lambda} n : nat.\, \textbf{Elim}(n; nat; P)\, \{pz, ps\}$$

which has the type $\boldsymbol{\Pi} P : nat \to \texttt{Prop}.\, (P\, Z) \to (\boldsymbol{\Pi} x : nat.\, P\, x \to P\, (S\, x)) \to \boldsymbol{\Pi} n : nat.\, P\, n$.

Judgmental equality. Figure 3 depicts an excerpt of the rules for judgemental equality. The rules BETA and ETA correspond to β and η equivalence. In this figure, we have elided the rules that specify that judgemental equality is an equivalence relation. The rules DELTA, ZETA and IOTA, respectively corresponding to unfolding definitions, expansion of let-ins and simplification of eliminators are also elided in Figure 3. The rule IOTA basically states that when the term being eliminated is a constructor c applied to certain values, then the result of elimination is judgementally equal to the corresponding case-eliminator f_c applied to the arguments of the constructor where (mutually) recursive arguments are appropriately eliminated. See Timany and Sozeau [19] for details. Note that the equivalence of the judgmental equality presentation and the implementation of definitional equality by conversion (as implemented in CoQ) is a tricky issue and it is still an open problem to formally show equivalence for a system with cumulativity [14], we leave this to future work.

Conversion/Cumulativity. Figure 4 shows an excerpt of conversion/cumulativity rules. The core of these rules is the rule CUM. It states that whenever a term t has type A and the conversion/cumulativity relation $A \preceq B$ holds, then t also has type B. The rule EQ-CUM says that two judgementally equal (convertible) types M and M' are in conversion/cumulativity

$$\text{Prop-in-Type} \qquad \frac{\qquad}{\Gamma \vdash \mathtt{Prop} \preceq \mathtt{Type}_i}$$

$$\text{Cum-Type} \qquad \frac{i \leq j}{\Gamma \vdash \mathtt{Type}_i \preceq \mathtt{Type}_j}$$

$$\text{Cum-Prod} \qquad \frac{\Gamma \vdash A_1 \simeq B_1 : s \qquad \Gamma, x : A_1 \vdash A_2 \preceq B_2}{\Gamma \vdash \mathbf{\Pi}x : A_1.\, A_2 \preceq \mathbf{\Pi}x : B_1.\, B_2}$$

$$\text{Cum} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \preceq B}{\Gamma \vdash t : B}$$

$$\text{Eq-Cum} \qquad \frac{\Gamma \vdash M \simeq M' : s}{\Gamma \vdash M \preceq M'}$$

Figure 4 An excerpt of conversion and cumulativity rules of pCIC.

relation $M \preceq M'$. The rules Prop-in-Type and Cum-Type specify the order on the hierarchy of sorts. The rule Cum-Prod states the conditions for conversion/cumulativity between two (dependent) function types. Note that in this rule, Π-types are *not* contravariant w.r.t. their domain. This is also the case in Coq. This condition is crucial for the construction of our set-theoretic model, since set-theoretic functions (*i.e.*, functional relations) are not contravariant.

3 Universes in Coq and pCIC

In the system that we have presented in this section, and for most of this paper, universe levels, *e.g.*, i in $\mathtt{Type}_i$, are explicitly specified. However, Coq enjoys a feature known as *typical ambiguity*. That is, users need not write universe levels explicitly; these are inferred by Coq. The idea here is that it suffices that there are universe levels, that can be placed in the appropriate place in the code, so that the code makes sense and respects consistent universe constraints. From a derivation with a consistent set of universe constraints one can always derive a pCIC derivation, using a valuation of the floating universe variables into the $\mathbb{U}_0 \ldots \mathbb{U}_n$ universes. This is exactly what is guaranteed using global universes and a global set of constraints on universe variables. In this sense the system pCIC as briefly discussed above forms a basis for Coq.

Universe polymorphism [15] extends Coq so that constructions can be made universe-polymorphic, *i.e.*, parameterized by some universe variables, following Harper and Pollack's seminal work [8]. That is, each universe-polymorphic definition will carry a context of universes together with a local set of constraints. The idea here is that any instantiation of a universe-polymorphic construction with universe levels that satisfy the local constraints is an acceptable one. In the implementation of conversion, universe levels only play a role when comparing two sorts or two polymorphic constants, inductives or constructors. In the kernel of Coq, only checking of the constraints is involved, they are hence global to a whole term type-checking process. The system is justified by a translation to pCIC as well, making "virtual" copies of every instance of universe-polymorphic constants and inductive types.

In this section we discuss these two features and how they treat inductive definitions. For the rest of this paper we will consider the systems pCIC and its extension pCuIC without either typical ambiguity or universe polymorphism. When describing the system pCuIC we will consider how changes to the base theory allows a different treatment of universe-polymorphic inductive types compared to pCIC.

Typical ambiguity, global algebraic universes and template polymorphism.   The user can only specify $\mathtt{Prop}$, $\mathtt{Set}$ or $\mathtt{Type}$. This is done by considering a collection of global algebraic universes (as opposed to local ones in universe-polymorphic constructions as we will see).

These universes are generated from the carrier set $\{\texttt{Set}\} \cup \{\mathbb{U}_\ell, | \ell \in \mathcal{L}\}$ for some countably infinite set of labels $\mathcal{L}$ (a.k.a. *levels*) with the operations max and successor ($+1$) (constructing *algebraic* universes).[2] Each use of the sort $\texttt{Type}$ is replaced with some $\texttt{Type}_{\mathbb{U}_\ell}$ for some fresh universe level ℓ. A global *consistent* set of constraints on the universe levels is kept at all times. When CoQ type checks a construction, it may add some constraints to this set. If adding a constraint would render the constraints inconsistent then the definition at hand is rejected with a *universe inconsistency* error. Let us consider the example of lists in CoQ[3].

```
Inductive list (A : Type@{𝕌ₗ}) : Type@{𝕌ₗ} :=
| nil : list A | cons : A → list A → list A. (\[* constraint added : 𝕌ₗ > Set *)
```

When CoQ processes the inductive definition of lists above, one constraint about $\mathbb{U}_\ell$ is added to the set of constraints, enforcing $\texttt{Set} < \ell$, as ℓ is global. The following set of constraints are added with the following definitions:

```
Definition nat_list := list nat.
 (\]* constraint added : 𝕌ₗ ≥ Set, already implied *)
Definition Set_list := list Set.
 (\[* constraint added : 𝕌ₗ > Set, already implied *)
Definition Type_list := list Type.
 (\]* constraint added : 𝕌ₗ > 𝕌ₗ' for some fresh 𝕌ₗ' for the occurrence o\[f Type *)
```

Template Polymorphism. Template polymorphism is a simple form of universe polymorphism for *non-universe-polymorphic* inductive types. It only applies to inductive types whose sort contain levels that appear *only* in one of their parameters and nowhere else in that inductive type. A prime example is the definition of `list` above. The sort of the inductive type appears only in the type of the only parameter. In case template polymorphism applies, different instantiations of the inductive types with different arguments for parameters can have different types. For instance, the terms above have different types:

```
Check (list nat). (* list nat : Set *)
Check (list Set). (* list Set : Type@{Set+1} *)
```

Here $\texttt{Type}\texttt{\backslash]@\{\mathbb{U}\}}$ is CoQ syntax for $\texttt{Type}_{\mathbb{U}}$. This feature is very important for reusability of the basic constructions such as lists. Crucially, template polymorphism considers two instances of a template-polymorphic inductive type convertible, whenever they are applied to arguments that are convertible, regardless of the universe in which these arguments are considered. That is, the following CoQ code type checks.

```
Universe i j. Constraint i < j.
Definition list_eq : list (nat : Type\[@{i}) = list (nat : Type\]@{j}) := eq_refl.
```

Universe polymorphism in pCIC and inductive types. The system pCIC has been extended with universe polymorphism [15]. This allows for definitions to be parameterized by universe levels. The essential idea here is that instead of declaring global universes for every occurrence of $\texttt{Type}$ in constructions, we use *local* universe levels (always $\geq \texttt{Set}$, which we omit in local constraints). That is, each universe-polymorphic construction carries with itself a context

---

[2] In CoQ, the sort $\texttt{Prop}$ is treated in a special way. In particular, $\texttt{Prop}$ is never unified with a universe $\texttt{Type}_{\mathbb{U}_\ell}$ for any algebraic universe $\mathbb{U}_\ell$.

[3] Here we show algebraic universes for the sake of clarity. These neither need to be written by the user nor are visible unless explicitly asked for. From now on, we will freely mention universe levels and constraints for presentation purposes but they can all be omitted.

of universe variables for universes that appear in the type and body of the construction together with a set of local universe constraints. These constraints may also mention global universe variables. This could happen in cases where the universe-polymorphic construction mentions universe-monomorphic constructions.

This feature allows us to define universe-polymorphic inductive types. The prime example of this is the polymorphic definition of categories:

```
Record Category@{i j} :=
 { Obj : Type\[@{i}; Hom : Obj → Obj → Type@{j}; ... }. (* local constraints: ∅ *)
```

This also allows us to define the category of (relatively small) categories as follows:[4]

```
Definition Cat@{i j k l} : Category@{i j} :=
 { Obj : Category@{k l}; ... }. (* local constr.: {k < i, l < i, k ≤ j, l ≤ j} *)
```

See Timany and Jacobs [18] for more details on using universe levels and constraints of Coq to represent (relative) smallness and largeness in category theory.

Note that the construction above, of the category of (relatively small) categories, could not be done in a similar way with a universe-monomorphic definition of category. This is because, the constraint $k < i$ would be translated to $\mathbb{U} < \mathbb{U}$ for some algebraic universe $\mathbb{U}$ that is taken to stand for the type of objects of categories. This would immediately make the global set of universe constraints inconsistent and thus the definition of category of categories would be rejected with a universe inconsistency error. Also notice that the universe-monomorphic version of the type `Category` is *not* template-polymorphic as the universe levels in the sort appear in the *constructor* of the type, and not only in its parameters and type.

Universe polymorphism treats inductive types at different universe levels as different types with no relation between them. This means that, in order to have a subtyping/cumulativity relation between two inductive types it requires the two instances to be at the exact same level. That is, for the subtyping relation `Category@{i j}` $\preceq$ `Category@{i' j'}` to hold it is required that $i = i'$ and $j = j'$. This means, among other things, that the category of categories defined above is not the category of all categories that are at most as large as k and l but those categories that are exactly at the level k and l.

This is not only about small and large objects like categories. Let `A` : `Type@{i}` be a type, obviously, `A` : `Type@{j}`, for any $j > i$. However, for the universe-polymorphic definition of lists, uplist, the types `uplist@{i}` (`A` : `Type@{i}`) and `uplist@{j}` (`A` : `Type@{j}`) are neither judgementally equal nor does the expected subtyping relation hold. In other words, the following Coq code will be accepted by Coq, *i.e.*, the reflexivity tactic will fail.[3]

```
Polymorphic Inductive uplist@{k} (A : Type@{k}) : Type@{k} :=
| upnil : uplist A | upcons : A → uplist A → uplist A.
Universe i j. Constraint i < j.
Lemma uplist_eq : uplist@{i} (nat : Type\]@{i}) = uplist@{j} (nat : Type\[@{j}).
 Fail reflexivity.
Abort.
```

As we discussed and demonstrated earlier, a similar equality with universe-monomorphic definition of lists does indeed hold. Note that the manually added constraint, `Constraint i < j`, is crucial here as otherwise the `reflexivity` tactic would succeed and Coq would silently equate universe levels i and j.

---

[4] There can be some other local constraints that we have omitted given rise to by mixing of universe-polymorphic and universe-monomorphic constructions, *e.g.*, if the definition of categories or **Cat** uses some universe-monomorphic definitions from the standrad library of Coq.

4 Predicative calculus of cumulative inductive constructions (pCuIC)

The system pCuIC extends the system pCIC by adding support for cumulativity between inductive types. This allows for different instances of a polymorphic inductive definition to be treated as subtypes of some other instances of the same inductive type under certain conditions.

The intuitive definition.   The intuitive idea for subtyping of inductive types is that an inductive type I is a subtype of another inductive type I' if they have the same *shape*, *i.e.*, the same number of parameters, indices and constructors, and corresponding constructors take the same number of arguments. Furthermore, it should be the case that every corresponding index (note that these do not include parameters) and every corresponding argument of every corresponding constructor have the expected subtyping relation (the one from I is a subtype of the one from I', *i.e.*, covariance) and also that corresponding constructors have the same end result type. One crucial point here is that we *only* compare inductive types if they are fully applied, *i.e.*, there are values applied for every parameter and index. This is because the cumulativity relation is only defined for types and not general arities.

Put more succinctly, given a term of type I applied to parameters and indices, it can be destructed and then reconstructed using the corresponding constructor of I', *i.e.*, terms of type I can be lifted to terms of type I' using identity coercions. Note that we do not consider parameters of the inductive types in question. This is because parameters of inductive types are basically forming different families of inductive types. For instance, the type `list A` and `list B` are two different families of inductive types. Not considering parameters allows our cumulativity relation for universe-polymorphic inductive types to mimic the behavior of template-polymorphic inductive types where the type of lists of a certain type are considered judgementally equal regardless of which universe level the type in question is considered to be in. Consider the following examples:

Example: categories.   The type `Category` being a record is an inductive type with a single constructor. In this case, there are no parameters or indices. The single constructors are constructing the same end result, *i.e.*, `Category`. As a result, in order to have the expected subtyping relation between `Category@{i j}` $\preceq$ `Category@{i' j'}`, i ≤ i' and j ≤ j', we need to have that these constraints suffice to show that every argument of the constructor of `Category@{i j}` is a subtype of the corresponding argument of the constructor of `Category@{i' j'}`. Note that it is only the first two arguments of the constructors that differ between these two types. The rest of the arguments, *e.g.*, composition of morphisms, associativity of composition, etc., are identical in both types. Hence, we only need to have the subtyping relations [5] $\mathrm{Type}_i \preceq \mathrm{Type}_{i'}$ and $Obj \to Obj \to \mathrm{Type}_j \preceq Obj \to Obj \to \mathrm{Type}_{j'}$ to hold and they do hold.

Example: lists.   The type of lists has a single parameter and no index, also notice that the universe level i in `list@{i}` does not appear in any of the two constructors. Hence, the subtyping relation `list@{i} A` $\preceq$ `list@{j} A` holds for any type `A` regardless of the relation between i and j.

---

[5] For the sake of clarity we have omitted the context under which these cumulativity relations need to hold.

Assuming $\mathcal{D} \equiv \mathsf{Ind}_n \{\Delta_I := \Delta_C\}$ and $\mathcal{D}' \equiv \mathsf{Ind}_n \{\Delta_I' := \Delta_C'\}$ we have:

IND-LEQ

$$
\dfrac{
\begin{array}{c}
\mathcal{D} \in \Gamma \qquad \mathcal{D}' \in \Gamma \qquad \mathsf{dom}(\Delta_I) = \mathsf{dom}(\Delta_I') \qquad \mathsf{dom}(\Delta_C) = \mathsf{dom}(\Delta_C') \\[4pt]
\Big[\Delta_I(d) \equiv \vec{p} : \vec{P}.\mathbf{\Pi}\vec{z} : \vec{V}.s \qquad \Delta_I'(d) \equiv \vec{p} : \vec{P}'.\mathbf{\Pi}\vec{z} : \vec{V}'.s' \qquad \Gamma, \vec{p} : \vec{P} \vdash \vec{V} \preceq \vec{V}' \\[4pt]
\Big(\Delta_C(c) \equiv \mathbf{\Pi}\vec{p} : \vec{P}.\,\mathbf{\Pi}\vec{x} : \vec{U}.d\;\vec{u} \qquad \Delta_C'(c) \equiv \mathbf{\Pi}\vec{p} : \vec{P}'.\,\mathbf{\Pi}\vec{x} : \vec{U}'.d\;\vec{u}' \qquad \Gamma, \vec{p} : \vec{P} \vdash \vec{U} \preceq \vec{U}' \\[4pt]
\Gamma, \vec{p} : \vec{P}, \vec{x} : \vec{U} \vdash \vec{u} \simeq \vec{u}' : \vec{P}', \vec{V}' \quad \text{ for } c \in \mathsf{Constrs}(\Delta_C, d) \Big) \quad \text{ for } d \in \mathsf{dom}(\Delta_I) \Big]
\end{array}
}{
\Gamma \vdash \mathcal{D} \preceq^\dagger \mathcal{D}'
}
$$

C-IND
$$
\dfrac{\Gamma \vdash \mathcal{D} \preceq^\dagger \mathcal{D}' \qquad \Gamma \vdash \mathcal{D}.d\;\vec{a} : s \qquad \Gamma \vdash \mathcal{D}'.d\;\vec{a} : s'}{\Gamma \vdash \mathcal{D}.d\;\vec{a} \preceq \mathcal{D}'.d\;\vec{a}}
$$

🟧 **Figure 5** Cumulativity for inductive types.

IND-EQ
$$
\dfrac{\Gamma \vdash \mathcal{D}.d\;\vec{a} \preceq \mathcal{D}'.d\;\vec{a} \qquad \Gamma \vdash \mathcal{D}'.d\;\vec{a} \preceq \mathcal{D}.d\;\vec{a} \qquad \Gamma \vdash \mathcal{D}.d\;\vec{a} : s \qquad \Gamma \vdash \mathcal{D}'.d\;\vec{a} : s}{\Gamma \vdash \mathcal{D}.d\;\vec{a} \simeq \mathcal{D}'.d\;\vec{a} : s}
$$

Assuming $\Gamma \vdash \mathcal{D}.c\;\vec{m} : \mathcal{D}.d\;\vec{a}$ and $\Gamma \vdash \mathcal{D}'.c\;\vec{m} : \mathcal{D}'.d\;\vec{a}$ we have :

CONSTR-EQ-L
$$
\dfrac{\Gamma \vdash \mathcal{D}'.d\;\vec{a} \preceq \mathcal{D}.d\;\vec{a}}{\Gamma \vdash \mathcal{D}.c\;\vec{m} \simeq \mathcal{D}'.c\;\vec{m} : \mathcal{D}.d\;\vec{a}}
$$

CONSTR-EQ-R
$$
\dfrac{\Gamma \vdash \mathcal{D}.d\;\vec{a} \preceq \mathcal{D}'.d\;\vec{a}}{\Gamma \vdash \mathcal{D}.c\;\vec{m} \simeq \mathcal{D}'.c\;\vec{m} : \mathcal{D}'.d\;\vec{a}}
$$

🟧 **Figure 6** Judgemental equality for inductive types.

Figure 5 shows the typing rules for cumulativity of inductive types. The rule `C-Ind` describes the condition for subtyping of inductive types $\mathcal{D}.d\;\vec{a}$ and $\mathcal{D}'.d\;\vec{a}$. This subtyping relation holds if the two types are fully applied, that is, the applications are terms of some sort s and s' respectively. It is also required that the inductive blocks $\mathcal{D}$ and $\mathcal{D}'$ are related under the $\preceq^\dagger$ relation. The rule IND-LEQ is rather lengthy but it essentially states what we explained above intuitively. It says that the relation $\mathcal{D} \preceq^\dagger \mathcal{D}'$ holds if the two blocks are defining inductive types with the same names and constructors with the same names. It also requires that for every corresponding inductive type in these blocks, the corresponding indices, are in the expected subtyping relation; similarly for corresponding arguments of corresponding constructors. Furthermore, corresponding constructors need to construct judgementally equal results.

Judgemental equality of inductive types.   Figure 6 shows the typing rules for judgemental equality of inductive types and their constructors. The rule IND-EQ states that two inductive types are considered to be judgementally equal if they are in mutual cumulativity relations.

This, and the judgemental equality for constructors explained below, allow universe polymorphism to mimic the behavior of template polymorphism for monomorphic inductive types. For instance, as we saw types `list@{i} A` is a subtype of `list@{j} A` for any type `A` regardless of `i` and `j`. Hence, using the rule IND-EQ it follows that the two types `list@{i} A` and `list@{j} A` are judgementally equal. However, the conditions of judgemental equality of universe-polymorphic inductive types is much more general compared to the conditions

$$\llbracket \Gamma \vdash \mathtt{Prop} \rrbracket_{\gamma} \triangleq \{\emptyset, \{\emptyset\}\} \qquad \llbracket \Gamma \vdash \mathtt{Type}_i \rrbracket_{\gamma} \triangleq \mathcal{V}_{\kappa_i} \qquad \llbracket \Gamma \vdash t \, u \rrbracket_{\gamma} \triangleq \mathsf{App}(\llbracket \Gamma \vdash t \rrbracket_{\gamma}, \llbracket \Gamma \vdash u \rrbracket_{\gamma})$$

$$\llbracket \Gamma \vdash \mathbf{\Pi} x : A. \, B \rrbracket_{\gamma} \triangleq \left\{ \mathsf{Lam}(f) \middle| f : \Pi a \in \llbracket \Gamma \vdash A \rrbracket_{\gamma}. \, \llbracket \Gamma, x : A \vdash B \rrbracket_{\gamma, a} \right\}$$

$$\llbracket \Gamma \vdash \boldsymbol{\lambda} x : A. \, t \rrbracket_{\gamma} \triangleq \mathsf{Lam}\left(\left\{ (a, \llbracket \Gamma, x : A \vdash t \rrbracket_{\gamma, a}) \middle| a \in \llbracket \Gamma \vdash A \rrbracket_{\gamma} \right\} \right)$$

■ **Figure 7** Excerpts of the model.

for template polymorphism to apply. Template polymorphism simply does not apply as soon as the universe in the sort is mentioned in any of the constructors. According to the rule IND-EQ, in order to get that the two types `Category@{i j}` and `Category@{i' j'}` are judgementally equal it is required that `i = i'` and `j = j'` as expected.

Judgemental equality of constructors. The rules CONSTR-EQ-L and CONSTR-EQ-R specify judgemental equality of constructors of inductive types in cumulativity relation. Let $\mathcal{D}.d \ \vec{a}$ and $\mathcal{D}'.d \ \vec{a}$ be two inductive types in the cumulativity relation $\mathcal{D}.d \ \vec{a} \preceq \mathcal{D}'.d \ \vec{a}$. Furthermore, let c be a constructor of the inductive blocks $\mathcal{D}$ and $\mathcal{D}'$ and $\vec{m}$ be terms such that $\mathcal{D}.c \ \vec{m}$ has type $\mathcal{D}.d \ \vec{a}$ and $\mathcal{D}'.c \ \vec{m}$ has type $\mathcal{D}'.d \ \vec{a}$. In this case, the rules CONSTR-EQ-L and CONSTR-EQ-R specify that $\mathcal{D}.c \ \vec{m}$ and $\mathcal{D}'.c \ \vec{m}$ are judgementally equal *at the highest* of the two types $\mathcal{D}.d \ \vec{a}$ and $\mathcal{D}'.d \ \vec{a}$.

This is another behavior of template polymorphism that the rules CONSTR-EQ-L and CONSTR-EQ-R allow us to mimic. For instance, consider the monomorphic and template-polymorphic inductive type of lists defined above. Template polymorphism of list implies that, *e.g.*, the empty list (the constructor `nil`) for the type of lists of a type A are judgementally equal regardless of the sort that A is in. That is, we have `nil (A : Type@{i})` $\simeq$ `nil (A : Type@{j})` regardless of `i` and `j`. Using the rules CONSTR-EQ-L and CONSTR-EQ-R we can achieve a similar result for the universe-polymorphic and inductive type of lists `uplist` defined above. These rules imply that `upnil@{i} A` $\simeq$ `upnil@{j} A` for any type `A` regardless of `i` and `j`.

5 Consistency

We establish the consistency of pCuIC by constructing a set theoretic model for the theory inspired by the model constructed by Lee and Werner [10]. We use our model to show (using relative consistency) that there are types that are not inhabited in the system. In fact, the model of Lee and Werner [10] does support cumulativity of inductive types. However, it is not suitable for showing consistency as it relies on the normalization of the body of fixpoints (structural recursion in COQ) for interpreting them. Furthermore, we work in ZFC set theory and use the axiom of choice *only* to show that the interpretation of inductive types constructed through fixpoints does indeed belong to the interpretation of the sort of the inductive type. Lee and Werner [10] work in ZF (with suitable cardinals, similarly to what we have assumed below) but we were not able to find a proof of this aspect of correctness of their interpretation of inductive types. See our extended technical appendix [19] for details.

The model. Here, we briefly present the most important parts of the model (see our extended technical appendix [19] for more details). We construct our set theoretic model in ZFC together with the axiom that there is a strictly increasing sequence of uncountable strongly inaccessible cardinals: $\kappa_0, \kappa_1, \ldots$ with $\kappa_0 > \omega$. Universe $\mathtt{Type}_i$ is interpreted as set theoretic (von Neumann) universes $\mathcal{V}_{\kappa_i}$ [5]. It is well-known [5] that the von Neumann universe

$\mathcal{V}_\kappa$ is a model of ZFC for any uncountable strongly inaccessible cardinal κ. We interpret the sort `Prop` as the set $\{\emptyset, \{\emptyset\}\}$. Figure 7 shows excerpts of our model of PCuIC. Interpretation of inductive types and eliminators are discussed below. We write $A\downarrow$ for well-definedness of the object A. We write $\Pi a \in A.\, B(a)$ for dependent set theoretic functions: $\Pi a \in A.\, B(a) \triangleq \left\{ f \in \left(\bigcup_{a \in A} B(a) \right)^A \middle| \forall a \in A.\, f(a) \in B(a) \right\}$. Here `Lam` and `App` are respectively functions that trace-encode a set-theoretic function and evaluate a trace encoded functions. Trace encoding is a standard technique [2] for set-theoretic representation of functions in a type theory with a proof-irrelevant universe (`Prop` in our case) which is a sub-type of another non-proof-irrelevant universe (`Prop` $\preceq$ `Type`$_i$ in our case).

Modeling inductive types and eliminators.   The basic idea of the interpretation of inductive types, constructors and eliminators is straightforward. However, the general presentation of the construction is lengthy and involves arguments regarding the general shape of inductive types. In particular, the strict positivity condition plays a crucial role. Here, we present the general idea and give some examples. Further details are available in Timany and Sozeau [19]. Following Lee and Werner [10], who follow Dybjer [6] and Aczel [2], we use inductive definitions (in set theory) constructed through rule sets to model inductive types. Here, we give a very short account of *rule sets* for inductive definitions. For further details refer to Aczel [1]. A rule set is a set of rules. A pair (A, a) is a rule based on a set U where $A \subseteq U$ is the set of premises and $a \in U$ is the conclusion. We write $\frac{A}{a}$ for a rule (A, a). The fixpoint $\mathcal{I}(\Phi)$ of a rule set Φ is the smallest set X such that for any rule $\frac{A}{a}$ if $A \subseteq X$ then $a \in X$. Every rule set has a fixpoint [1].

The idea here is to construct a rule set for the whole inductive block. For each collection of arguments that can possibly be applied to a constructor we add a rule to the rule set. The premises of the rule requires that all (mutually) recursive arguments are in the fixpoint. We define the interpretation of individual inductive types based on this fixpoint. Let $\mathcal{D} \equiv \mathbf{Ind}_0\{nat : \texttt{Set} := Z : nat, S : nat \to nat\}$ be the inductive block for inductive definition of natural numbers. The rule set for this inductive block is as follows:

$$\Phi_\mathcal{D} \triangleq \left\{ \frac{\emptyset}{\langle 0; \mathsf{nil}; \mathsf{nil}; \langle 0; \mathsf{nil}\rangle\rangle} \right\} \cup \left\{ \frac{\{\langle 0; \mathsf{nil}; \mathsf{nil}; a\rangle\}}{\langle 0; \mathsf{nil}; \mathsf{nil}; \langle 1; a\rangle\rangle} \middle| a \in \mathcal{V}\kappa_0 \right\}$$

The rule corresponding to Z has no premise as Z takes no recursive argument. This rule concludes that the term $\langle 0; \mathsf{nil}\rangle$, *i.e.*, zero[th] constructor applied to `nil` arguments is a term of zero[th] type with `nil` as both parameters and indices. The rules corresponding to S state that $\langle 1; a\rangle$ is an element of the zero[th] type if a is. Based on this fixpoint we define the semantics of natural numbers, $\llbracket \cdot \vdash \mathcal{D}.nat \rrbracket_\mathsf{nil} \triangleq \{\langle k; \vec{a}\rangle | \langle 0; \mathsf{nil}; \mathsf{nil}; \langle k; \vec{a}\rangle\rangle \in \mathcal{I}(\Phi_\mathcal{D})\}$, zero, $\llbracket \cdot \vdash \mathcal{D}.Z \rrbracket_\mathsf{nil} \triangleq \langle 0; \mathsf{nil}\rangle$ and successor, $\llbracket \cdot \vdash \mathcal{D}.S \rrbracket_\mathsf{nil} \triangleq \mathsf{Lam}\left(\{(a, \langle 1; a\rangle) | a \in \llbracket \cdot \vdash \mathcal{D}.nat \rrbracket_\mathsf{nil}\}\right)$.

Interpreting eliminators.   We use rule sets to also define the interpretation of eliminators. For each constructor applied to a sequence of arguments we add a rule to the rule set. This rule states that the result of elimination is exactly the result of applying the corresponding case eliminator where the result of elimination of (mutually) recursive arguments are taken as arbitrary sets. The premise requires that each set taken as elimination of a (mutually) recursive argument is mapped correctly in the fixpoint. We define the interpretation of elimination of a term t of an inductive type as the set a if a is the unique set such that the pair $(\llbracket t \rrbracket, a)$ is in the fixpoint of the elimination. Assume we have sets r, rz and rs such that $r, rz, rs \in \llbracket \Gamma \rrbracket$ where $\Gamma = Q : nat \to \texttt{Type}_i, qz : Q\, Z, qs : \mathbf{\Pi}x : nat.\, Q\, x \to Q\, (S\, x)$. The rule

set for the elimination of natural numbers is as follows:

$$\Phi_{ELB} \triangleq \left\{ \frac{\emptyset}{(\langle 0; \mathsf{nil}\rangle\,, rz)} \right\} \cup \left\{ \frac{\{(a,b)\}}{(\langle 1; a\rangle\,, \overrightarrow{\mathsf{App}}(rs, a, b))} \middle| \begin{array}{l} a \in [\![\Gamma \vdash \mathcal{D}.nat]\!]_{r,rz,rs}\,, \\ b \in \mathcal{V}\kappa_i \end{array} \right\}$$

We define the interpretation of elimination of the term n as a if a is the unique set such that the pair $([\![\Gamma \vdash n]\!]_{r,rz,rs}\,, a) \in \mathcal{I}(\Phi_{ELB})$.

Soundness theorem and consistency.

▶ **Theorem 1** (Soundness of the model)**.**

1. *If* $\mathcal{WF}(\Gamma)$ *then* $[\![\Gamma]\!]\!\downarrow$
2. *If* $\Gamma \vdash t : A$ *then* $[\![\Gamma]\!]\!\downarrow$ *and for any* $\gamma \in [\![\Gamma]\!]$ *we have* $[\![\Gamma \vdash t]\!]_\gamma\!\downarrow$, $[\![\Gamma \vdash A]\!]_\gamma\!\downarrow$ *and* $[\![\Gamma \vdash t]\!]_\gamma \in [\![\Gamma \vdash A]\!]_\gamma$
3. *If* $\Gamma \vdash t \simeq t' : A$ *then* $[\![\Gamma \vdash t]\!]_\gamma\!\downarrow$, $[\![\Gamma \vdash t']\!]_\gamma\!\downarrow$, $[\![\Gamma \vdash A]\!]_\gamma\!\downarrow$ *and* $[\![\Gamma \vdash t]\!]_\gamma = [\![\Gamma \vdash t']\!]_\gamma \in [\![\Gamma \vdash A]\!]_\gamma$
4. *If* $\Gamma \vdash A \preceq B$ *then* $[\![\Gamma \vdash A]\!]_\gamma\!\downarrow$, $[\![\Gamma \vdash B]\!]_\gamma\!\downarrow$ *and* $[\![\Gamma \vdash A]\!]_\gamma \subseteq [\![\Gamma \vdash B]\!]_\gamma$

Proof. By mutual induction on the typing derivations. For C-IND we need to show that the interpretation of one inductive type is a subset of the interpretation of the other one. This follows from the fact that the arguments of constructors of the two types have the required subset relation (by induction hypothesis).The cases IND-EQ, CONSTR-EQ-L and CONSTR-EQ-R are trivial. ◀

▶ **Corollary 2** (Consistency of PCUIC)**.** *Let* s *be a sort, then, there* does not *exist any term* t *such that* $\cdot \vdash t : \mathbf{\Pi}x : s.\,x$.

Proof. If there were such a term t, by Theorem 1 we would have $[\![\cdot \vdash t]\!]_{\mathsf{nil}} \in [\![\cdot \vdash \mathbf{\Pi}x : s.\,x]\!]_{\mathsf{nil}}$. However, $[\![\cdot \vdash \mathbf{\Pi}x : s.\,x]\!]_{\mathsf{nil}} = \emptyset$. ◀

6 Coq implementation

We implemented the extension to PCIC, that are presented in this paper, in the COQ system, which is now available as of the stable 8.7 version of the system [16], documented[6] and even experimented with already in the UniMath library.[7]

From the user point of view, this adds a new optional flag on universe-polymorphic inductive types that computes the cumulativity relation for two arbitrary fresh instances of the inductive type that can be printed afterwards using the `Print` command. Cumulativity and conversion for the fully applied inductive type and its constructors is therefore modified to use the cumulativity constraints instead of forcing equalities everywhere as was done before, during unification, typechecking and conversion. As cumulativity is always potentially more relaxed than conversion, users can set this option in existing developments and maintain compatibility. Of course actually making use of the new feature is not backward-compatible.

---

[6] https://coq.inria.fr/distrib/current/refman/addendum/universe-polymorphism.html
[7] See the discussion on GitHub: https://github.com/UniMath/UniMath/issues/648

Impact on the Coq codebase.   The impact of this extension is relatively small as it involves mainly an extension of the data-structures representing the universes associated with polymorphic inductive types in the Coq kernel, and their use during the conversion test of Coq, which was already generic in the tests used for comparing polymorphic inductives and constructors. Note that we have not needed to adapt the two efficient *reduction* strategies of Coq, `vm_compute` and `native_compute`, as universes are irrelevant for reduction. A good chunk of changes involved cleanups of the kernel API for registering inductive declarations.

Performance.   When no inductive type is declared cumulative, the extension has no impact, as we tested on a large set of user contributions including the Mathematical Components and the Coq HoTT library (the common stress-tests for universes). When activated globally, we hit one case in the test-suite of Coq taken from the HoTT library where the computation of the subtyping relation for a given inductive blows up, due to conversion unfolding definitions to infer the subtyping constraints. In this case we know that the relation would be trivial (cumulativity collapses to equality), hence we were motivated to make the `Cumulative` flag optional. The performance is otherwise not affected, as far as we know.

7   Applications

In this section we briefly discuss two motivating applications that are made possible thanks to the new cumulativity feature for inductive types that we have presented here.

Yoneda embedding.   Each category `C : Category@{i j}` is equipped with a *hom*-functor, `Hom_func : C × C`^{op}` → Type_Cat@{j}`. Here `Type_Cat` is the category of types and functions, which plays the role of the *Set* category. It is expected that one could define the Yoneda embedding `Y(C)` as `Curry Hom_func` where `Curry` is the exponential transpose of the cartesian closed structure of the category of categories `Cat`. However, the cartesian closed version of `Cat@{i' j' k' l'}` has the constraints $k' = l' = j'$ and `Type_Cat@{j} : Category@{k j}` with the side constraint $j < k$. This means that `Type_Cat` is not an object of any cartesian closed version of `Cat` making it impossible to use `Curry` on `Hom_func`. See Timany and Jacobs [18] for a detailed discussion of this issue.

    Cumulativity of inductive types solves this issue. In pCuIC, `Type_Cat` is indeed an object of a cartesian closed version of `Cat` at some higher universe level allowing us to directly use exponential transpose to define the Yoneda embedding.

Syntactical models of type theories.   In [4], Boulier *et al.* advocate the study of syntactical models of type theory, that is models defined by definitional translations from a source type theory to a target type theory. A definitional translation of dependent type theory must preserve its conversion relation, which is known as "computational soundness" in proof theory in general. In pCIC and pCuIC, it must preserve the cumulativity relation.

    A most basic example of syntactical model is the "cross-bool" model, which interprets every type as the type itself crossed with booleans, *i.e.*, using a polymorphic pair type: $[\text{Type}_i] = (\text{Type}_i \times_{j,\text{Set}} \mathbb{B}, \text{true})$ where $i < j$ $[\![A]\!] = [A].1$

    Likewise, every term is interpreted as the term itself *plus a boolean*. This model can be used to show that type extensionality, hence univalence, is independent from CC_ω (*op. cit.*). However, this model does not scale to Coq's type theory as the cumulativity rule is not validated through the translation. Indeed to validate cumulativity one must have, assuming $i \leq k \wedge i < j \wedge k < l$: $[\![\text{Type}_i]\!] \leq [\![\text{Type}_k]\!] \triangleq (\text{Type}_i \times_{j,\text{Set}} \mathbb{B}, \text{true}).1 \leq (\text{Type}_k \times_{l,\text{Set}} \mathbb{B}, \text{true}).1$

This judgement holds only if $j = l$ and $i = k$ in PCIC, and is relaxed to only $i = k$ in PCuIC. The latter constraint is forced due to the appearance of the types as parameters of the pair type. We can go one step further and define a specialized inductive type:

```
Inductive TyInterp@{i j | i < j} : Type@{j} := { T : Type@{i}; b : bool }.
```

The subtyping constraints on `TyInterp` will only require that $i \leq k$, as assumed! Note also that template polymorphism would not help here as the type is not a parameter anymore.

8 Future and related work

Moving from template polymorphism to universe polymorphism. One motivation for this extension to explain the so-called "template-polymorphic" inductive types of Coq in terms of cumulative universe-polymorphic inductive types. This puts the system on a clean and solid theoretical ground. Furthermore, we would like to switch the standard library of Coq to full universe polymorphism. Making the template-polymorphic inductives, in the standard library, interact with universe-polymorphic code is prone to introduce universe inconsistencies; the two systems work in quite different ways. Hence, we have tried to set universe polymorphism on everywhere.

Our experiments are encouraging but not without issues. We are able to make the basic inductive types of the standard library cumulative universe-polymorphic, and all constants polymorphic (except in a few files devoted to the formalization of paradoxes). We found that the relaxed rule on constructors was necessary in some cases, this is a case where practice met theory: our model construction justified the required relaxation for these examples.

However, we hit an orthogonal problem with the definitions of modules and module types, used to formalize the number and finite map and set libraries for example, where definitions drastically change meaning when interpreted in universe-polymorphic mode. Indeed, when a module parameter `A :` `Type` is declared in monomorphic mode, one gets a floating universe, *i.e.*, it is elaborated to $A : \text{Type}_\ell$ for some global universe ℓ. In universe polymorphism mode it is elaborated to $A@\{\ell\} : \text{Type}_\ell$ instead, which can only be instantiated by `Prop` and types in `Set`, at the bottom of the hierarchy. The only way to fix this is to add user annotations in the files to switch between monomorphic and polymorphic mode, which is work-in-progress.

We believe that our extension to PCIC maintains strong normalization and that the model constructed by Barras [3] could be easily extended to support our added rules.

Related Work. We are not aware of any other system providing cumulativity on inductive types, neither MATITA nor LEAN, the closest cousins of Coq, implement cumulativity. They prefer the algebraic presentation of universes that is also used in AGDA and where explicit lifting functions must be defined between different instances of polymorphic inductive types. In [11], McBride presents a proposal for internalizing "shifting" of universe-polymorphic constructions to higher universe levels akin to an explicit version of cumulativity that was further studied by Rouhling [13], but parameterized inductive types are not considered there.

9 Conclusion

We have presented a sound extension of the predicative calculus of inductive constructions with cumulative inductive types, which allows to equip cumulative universe-polymorphic inductive types with definitional equalities and reasoning principles that are closer to the "informal" mathematical practice. Our system is implemented in the Coq proof assistant and is justified by a model construction in ZFC set theory. We hope to make this feature

more useful and applicable once we resolve the remaining, orthogonal issue with the module system, allowing users of the standard library of Coq to profit from it as well.

─── **References** ───

1   Peter Aczel. An Introduction to Inductive Definitions. *Studies in Logic and the Foundations of Mathematics*, 90:739–782, 1977.

2   Peter Aczel. On Relating Type Theories and Set Theories. In Thorsten Altenkirch, Bernhard Reus, and Wolfgang Naraschewski, editors, *TYPES'98*, pages 1–18. Springer Berlin Heidelberg, 1999.

3   Bruno Barras. Semantical Investigation in Intuitionistic Set Theory and Type Theoris with Inductive Families, 2012. Habilitation thesis draft, University Paris Diderot – Paris 7.

4   Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. The Next 700 Syntactical Models of Type Theory. In *CPP 2017*, pages 182–194, Paris, France, 2017. `doi:10.1145/3018610.3018620`.

5   Frank R Drake. *Set theory : an introduction to large cardinals.* Studies in logic and the foundations of mathematics 76. North-Holland, Amsterdam, 1974.

6   Peter Dybjer. *Inductive Sets and Families in Martin-Löf's Type Theory and Their Set-Theoretic Semantics*, pages 280–306. Cambridge University Press, Cambridge, 1991.

7   Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

8   Robert Harper and Robert Pollack. Type checking, universe polymorphism, and typical ambiguity in the calculus of constructions draft. In J. Díaz and F. Orejas, editors, *TAPSOFT '89*, pages 241–256, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg.

9   Antonius JC Hurkens. A simplification of Girard's paradox. In *International Conference on Typed Lambda Calculi and Applications*, pages 266–278, Edinburgh, UK, 1995. Springer.

10   Gyesik Lee and Benjamin Werner. Proof-irrelevant model of CC with predicative induction and judgmental equality. *Logical Methods in Computer Science*, 7(4), 2011. `doi:10.2168/LMCS-7(4:5)2011`.

11   Conor McBride. Universe hierarchies, 2015. Blog post.

12   C. Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supérieur.* Habilitation à diriger les recherches, Université Claude Bernard Lyon I, 1996.

13   Damien Rouhling. Dependently typed lambda calculus with a lifting operator. Technical report, ENS Lyon, May-August 2014. Internship report.

14   Vincent Siles and Hugo Herbelin. Pure type system conversion is always typable. *J. Funct. Program.*, 22(2):153–180, 2012. `doi:10.1017/S0956796812000044`.

15   Matthieu Sozeau and Nicolas Tabareau. Universe Polymorphism in Coq. In *Interactive Theorem Proving 2014*, pages 499–514, Vienna, Austria, 2014. Springer. `doi:10.1007/978-3-319-08970-6_32`.

16   The Coq Development Team. The Coq Proof Assistant, version 8.7.1, dec 2017. `doi:10.5281/zenodo.1133970`.

17   Amin Timany and Bart Jacobs. First Steps Towards Cumulative Inductive Types in CIC. In *Theoretical Aspects of Computing*, pages 608–617, Cali, Colombia, 2015. Springer. `doi:10.1007/978-3-319-25150-9_36`.

18   Amin Timany and Bart Jacobs. Category Theory in Coq 8.5. In *Formal Structures for Computation and Deduction*, pages 30:1–30:18, Porto, Portugal, 2016. LIPIcs. `doi:10.4230/LIPIcs.FSCD.2016.30`.

19   Amin Timany and Matthieu Sozeau. Consistency of the Predicative Calculus of Cumulative Inductive Constructions (pCuIC). Research Report 9105, KU Leuven ; Inria Paris, 2017. URL: `https://hal.inria.fr/hal-01615123`.

Completion for Logically Constrained Rewriting

Sarah Winkler

Department of Computer Science, University of Innsbruck, Austria
sarah.winkler@uibk.ac.at
 https://orcid.org/0000-0001-8114-3107

Aart Middeldorp

Department of Computer Science, University of Innsbruck, Austria
aart.middeldorp@uibk.ac.at
 https://orcid.org/0000-0001-7366-8464

—————— Abstract ——————

We propose an abstract completion procedure for logically constrained term rewrite systems (LCTRSs). This procedure can be instantiated to both standard Knuth-Bendix completion and ordered completion for LCTRSs, and we present a succinct and uniform correctness proof. A prototype implementation illustrates the viability of the new completion approach.

1 Introduction

Rewriting in the presence of side constraints captures simplification processes in various areas, such as expression rewriting in compilers, theorem provers, or SMT solvers [10, 15, 17]. The imposed side constraints can often be expressed as logical formulas. *Logically constrained rewrite systems* [13] formalize this rewriting mechanism, admitting side constraints over an arbitrary first-order logic. Though their application for practical analysis tasks relies on satisfiability checks in the respective logic, thanks to the advent of powerful SMT solvers in the last decade LCTRSs are valuable in a wide range of areas, including program verification [7].

Often simplification procedures aim for unique results. *Knuth-Bendix completion* [11] thus poses a natural means to obtain a presentation of the rewrite system which is confluent and terminating, such that unique results are guaranteed. In particular, such a presentation can be used to decide the validity problem. Standard completion may fail if unorientable equations are encountered. To address this drawback, *ordered completion* was proposed by Bachmair, Dershowitz, and Plaisted [3]. This variant of completion never fails, at the price of the resulting system being only ground complete.

In this paper we propose an abstract inference system for completion of LCTRSs. This deduction scheme can be instantiated to both standard and ordered completion procedures, depending on the success condition satisfied by a run. To this end, we also state and prove a critical pair lemma for LCTRSs. Correctness proofs of completion procedures traditionally relied on proof orders. In contrast, we give proofs that exploit the more recent notion of

peak decreasingness [9] to show (ground) confluence, which is the key part of the proof. This approach permits a succinct and uniform proof for finite runs of both standard and ordered completion. It also shows how peak decreasingness does not only apply to ordered completion, but also extends to the considerably more intricate setting of constrained rewriting.

To ensure termination of the resulting rewrite system, we propose the notion of a *constrained reduction order*, which generalizes the recursive path order presented in [13]. Since any terminating LCTRS gives rise to such an order, the proposed completion procedures can also be implemented using termination tools instead of a fixed reduction order, a known approach in the unconstrained setting [19]. We outline an implementation within the tool Ctrl [14] and give examples that illustrate the practicality of our method.

Overview. The remainder of this paper is organized as follows. In Section 2 we summarize the relevant background. Section 3 is devoted to constrained reduction orders. To analyze peaks over LCTRSs, we prove a critical pair lemma in Section 4. Our inference system along with all proofs is presented in Section 5. Section 6 outlines our implementation in Ctrl before we conclude in Section 7. Due to space limitations, some proofs were moved to an appendix.

2 Preliminaries

We assume familiarity with the basic notions of term rewrite systems (TRSs) and completion [1, 2], but shortly recapitulate terminology and notation that we use in the remainder. In particular, we recall the notion of logically constrained rewriting as defined in [7, 13].

Terms. We assume a sorted signature $\mathcal{F} = \mathcal{F}_{\text{terms}} \cup \mathcal{F}_{\text{theory}}$. The set $\mathcal{T}(\mathcal{F}, \mathcal{V})$ denotes the terms over this signature. We assume a mapping $\mathcal{I}$ which assigns to every sort ι occurring in $\mathcal{F}_{\text{theory}}$ a carrier set $\mathcal{I}(\iota)$, and an interpretation $\mathcal{J}$ that assigns to every symbol $f \in \mathcal{F}_{\text{theory}}$ of sort $\iota_1 \times \cdots \times \iota_n \to \kappa$ a function $f_{\mathcal{J}} : \mathcal{I}(\iota_1) \times \cdots \times \mathcal{I}(\iota_n) \to \mathcal{I}(\kappa)$. Moreover, for every sort ι occurring in $\mathcal{F}_{\text{theory}}$ we assume a set $\mathcal{V}\mathsf{al}_\iota \subseteq \mathcal{F}_{\text{theory}}$ of value symbols, such that all $c \in \mathcal{V}\mathsf{al}_\iota$ are constants of sort ι and $\mathcal{J}$ constitutes a bijective mapping between $\mathcal{V}\mathsf{al}_\iota$ and $\mathcal{I}(\iota)$. Thus there exists a constant symbol for every value in the carrier set. The interpretation $\mathcal{J}$ naturally extends to an interpretation of ground terms, mapping ground terms to values:

$$[f(t_1, \ldots, t_n)]_{\mathcal{J}} = f_{\mathcal{I}}([t_1]_{\mathcal{J}}, \ldots, [t_n]_{\mathcal{J}})$$

Thus every ground term has a unique value. We demand that theory symbols and term symbols overlap only on values, i.e., $\mathcal{F}_{\text{terms}} \cap \mathcal{F}_{\text{theory}} \subseteq \mathcal{V}\mathsf{al}$ holds. A term in $\mathcal{T}(\mathcal{F}_{\text{theory}}, \mathcal{V})$ is called a *logical* term. Moreover we assume existence of a sort bool such that $\mathcal{I}(\mathsf{bool}) = \mathbb{B} = \{\top, \bot\}$, $\mathcal{V}\mathsf{al}_{\mathsf{bool}} = \{\mathsf{true}, \mathsf{false}\}$, $[\mathsf{true}]_{\mathcal{J}} = \top$, and $[\mathsf{false}]_{\mathcal{J}} = \bot$ hold. Logical terms of sort bool are called *constraints*. A constraint φ is *valid* if $[\varphi\gamma]_{\mathcal{J}} = \top$ for all substitutions γ such that $\gamma(x) \in \mathcal{V}\mathsf{al}$ for all $x \in \mathcal{V}\mathsf{ar}(\varphi)$.

Rewriting with Constraints. A *constrained equation* is a triple $\ell \approx r \ [\varphi]$ where $\ell, r \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ are of the same sort and φ is a constraint. If $\varphi = \mathsf{true}$ then the constraint is often omitted, and the equation denoted as $\ell \approx r$. Sometimes $s \simeq t$ is used to abbreviate "$s \approx t$ or $t \approx s$". A *constrained rewrite rule* is a constrained equation such that $\mathsf{root}(\ell) \in \mathcal{F}_{\text{terms}} \setminus \mathcal{F}_{\text{theory}}$ holds and which is denoted $\ell \to r \ [\varphi]$. For a set of constrained equations $\mathcal{E}$, we write $\mathcal{E}^{-1}$ for $\{v \approx u \ [\varphi] \mid u \approx v \ [\varphi] \in \mathcal{E}\}$ and $\mathcal{E}^{\pm}$ to denote $\mathcal{E} \cup \mathcal{E}^{-1}$. A set of constrained rewrite rules is called a *logically constrained rewrite system* (LCTRS for short). We now define rewriting using constrained equations. To this end, a substitution σ is said to *respect* a constraint φ if $\varphi\sigma$ is valid and $\sigma(x) \in \mathcal{V}\mathsf{al}$ for all $x \in \mathcal{V}\mathsf{ar}(\varphi)$.

▶ **Definition 1.** Let $\mathcal{E}$ be a set of constrained equations.

- A *calculation step* $s \to_{\mathsf{calc}} t$ satisfies $s = C[f(s_1, \ldots, s_n)]$ for some $f \in \mathcal{F}_{\text{theory}} \setminus \mathcal{V}\text{al}$, $t = C[u]$, $s_i \in \mathcal{V}\text{al}$ for all $1 \leqslant i \leqslant n$, and $u \in \mathcal{V}\text{al}$ is the value symbol of $[f(s_1, \ldots, s_n)]_{\mathcal{J}}$. In this case $f(x_1, \ldots, x_n) \to y \ [y = f(x_1, \ldots, x_n)]$ with $y \in \mathcal{V}$ is a *calculation rule*.
- A *rule step* $s \to_{\ell \approx r \ [\varphi]} t$ satisfies $s = C[\ell\sigma]$, $t = C[r\sigma]$, and σ respects φ.

We also write $\to_{\mathsf{rule}, \mathcal{E}}$ to refer to the relation $\{\to_\alpha\}_{\alpha \in \mathcal{E}}$, and denote $\to_{\mathsf{calc}} \cup \to_{\mathsf{rule}, \mathcal{E}}$ by $\to_{\mathcal{E}}$.

We sometimes write $\to_{p, \mathcal{E}}$ to indicate that the rewrite step takes place at position p. The subscript $\mathcal{E}$ is dropped if clear from the context. Moreover, the LCTRS $\mathcal{R}_{\mathsf{calc}}$ refers to the set of all calculation rules. In contrast to [13] we also use equations for rewriting, so we do not require that a rule step using $\ell \approx r \ [\varphi]$ with substitution σ satisfies $\sigma(x) \in \mathcal{V}\text{al}$ for all logical variables of the rule, i.e., also for $x \in \mathcal{V}\text{ar}(r) \setminus (\mathcal{V}\text{ar}(\ell) \cup \mathcal{V}\text{ar}(\varphi))$.

Note that $\to_{\mathsf{calc}}$ is terminating since calculation steps strictly reduce the number of non-value symbols in terms.

▶ **Example 2.** Consider the sort int (besides bool) and let $\mathcal{F}_{\text{theory}}$ consist of symbols $\cdot$, $+$, $-$, $\leqslant$, and $\geqslant$ as well as values n for all $n \in \mathbb{Z}$, with the usual interpretations on $\mathbb{Z}$. Let $\mathcal{F}_{\text{terms}} = \mathcal{V}\text{al} \cup \{\mathsf{fact}\}$. The LCTRS $\mathcal{R}$ consisting of the rules

$$\mathsf{fact}(x) \to 1 \quad [x \leqslant 0] \qquad\qquad \mathsf{fact}(x) \to \mathsf{fact}(x - 1) \cdot x \quad [x - 1 \geqslant 0]$$

admits the following rewrite steps:

$$
\begin{aligned}
\mathsf{fact}(2) &\to_{\mathsf{rule}} \mathsf{fact}(2 - 1) \cdot 2 & &&&(\text{as } 2 - 1 \geqslant 0 \text{ is valid}) \\
&\to_{\mathsf{calc}} \mathsf{fact}(1) \cdot 2 & \to_{\mathsf{rule}} &\ (\mathsf{fact}(1 - 1) \cdot 1) \cdot 2 & &(\text{as } 1 - 1 \geqslant 0 \text{ is valid}) \\
&\to_{\mathsf{calc}} (\mathsf{fact}(0) \cdot 1) \cdot 2 & \to_{\mathsf{rule}} &\ (1 \cdot 1) \cdot 2 & &(\text{as } 0 \leqslant 0 \text{ is valid}) \\
&\to_{\mathsf{calc}}^+ 2
\end{aligned}
$$

An LCTRS $\mathcal{R}$ is *terminating* if $\to_{\mathcal{R}}$ is well-founded, and *confluent* if $_{\mathcal{R}}^*\!\!\leftarrow \cdot \to_{\mathcal{R}}^* \subseteq \to_{\mathcal{R}}^* \cdot {}_{\mathcal{R}}^*\!\!\leftarrow$. We use *peak decreasingness* [9] as confluence criterion. An abstract rewrite system $\mathcal{A} = \langle A, \{\to_\alpha\}_{\alpha \in I} \rangle$ is *peak decreasing* if there exists a well-founded order $>$ on I such that for all $\alpha, \beta \in I$ the inclusion ${}_\alpha\!\leftarrow \cdot \to_\beta \ \subseteq \xleftrightarrow[\vee\alpha\beta]{*}$ holds. Here $\vee\alpha\beta$ denotes the set $\{\gamma \mid \alpha > \gamma \text{ or } \beta > \gamma\}$.

▶ **Lemma 3** ([9]). *Every peak decreasing ARS is confluent.*

Rewriting Constrained Terms. Logically constrained rewriting aims to rewrite unconstrained terms with constrained rules. However, for the sake of analysis, rewriting *constrained terms* is useful. In particular, our completion procedure will maintain sets of constrained equations, and rewrite constrained terms. We recall the relevant notions [7, 13].

A *constrained term* is a pair $s \ [\varphi]$ of a term s and a constraint φ. Two constrained terms $s \ [\varphi]$ and $t \ [\psi]$ are *equivalent*, denoted by $s \ [\varphi] \sim t \ [\psi]$, if for every substitution γ respecting φ there is some substitution δ that respects ψ such that $s\gamma = t\delta$, and vice versa. For example, $\mathsf{fact}(x) \cdot x \ [x = 1 \wedge x < y] \sim \mathsf{fact}(1) \cdot y \ [y > 0 \wedge y < 2]$ holds, but these terms are not equivalent to $\mathsf{fact}(x) \cdot y \ [x = y]$ or $\mathsf{fact}(1) \ [\mathsf{true}]$.

▶ **Definition 4.** Let $\mathcal{E}$ be a set of constrained equations.

- A *calculation step* $s \ [\varphi] \to_{\mathsf{calc}} t \ [\varphi \wedge x = f(s_1, \ldots, s_n)]$ satisfies $s = C[f(s_1, \ldots, s_n)]$ for some $f \in \mathcal{F}_{\text{theory}} \setminus \mathcal{F}_{\text{terms}}$ and $t = C[x]$ such that $s_1, \ldots, s_n \in \mathcal{V}\text{ar}(\varphi) \cup \mathcal{V}\text{al}$ and x is a fresh variable.
- A constraint rewrite rule $\alpha \colon \ell \to r \ [\psi]$ admits a *rule step* $s \ [\varphi] \to_\alpha t \ [\varphi]$ if φ is satisfiable, $s = C[\ell\sigma]$, $t = C[r\sigma]$, $\sigma(x) \in \mathcal{V}\text{al} \cup \mathcal{V}\text{ar}(\varphi)$ for all $x \in \mathcal{V}\text{ar}(\psi)$, and $\varphi \Rightarrow \psi\sigma$ is valid.

Given an LCTRS $\mathcal{E}$, we again write $\to_{\mathsf{rule},\,\mathcal{E}}$ for $\{\to_\alpha\}_{\alpha\in\mathcal{E}}$. The main rewrite relation $\to_\mathcal{E}$ on constrained terms is defined as $\sim\cdot(\to_{\mathsf{calc}}\cup\to_{\mathsf{rule},\,\mathcal{E}})\cdot\sim$.

▶ **Example 5.** Consider the LCTRS from Example 2, the constraint $\varphi = x \geqslant 1 \wedge y \geqslant 0$, and let z be a fresh variable. Then the following rewrite steps are possible:

$$\mathsf{fact}(x+y)\ [\varphi] \to_{\mathsf{rule}} \mathsf{fact}(x+y-1)\cdot(x+y)\ [\varphi]$$
$$\mathsf{fact}(x+y)\ [\varphi] \to_{\mathsf{calc}} \mathsf{fact}(z)\ [\varphi \wedge z = x+y]$$

The following key results relate rewriting on constrained terms to rewriting on unconstrained terms.

▶ **Lemma 6** ([13, Lemma 2]). *If $s\ [\varphi] \to_{\mathsf{rule}} t\ [\psi]$ and γ respects φ then $s\gamma \to_{\mathsf{rule}} t\gamma$. The two steps take place at the same positions.*

▶ **Lemma 7** ([7, Theorems 2.19 and 2.20]). *Suppose $s\ [\varphi] \to_{p,\mathcal{R}} t\ [\psi]$.*
1. *If γ respects φ then $s\gamma \to_{p,\mathcal{R}} t\delta$ for some substitution δ respecting ψ.*
2. *If δ respects ψ then $s\gamma \to_{p,\mathcal{R}} t\delta$ for some substitution γ respecting φ.*

Using a fresh binary term symbol $\langle\cdot,\cdot\rangle$, we show a slightly stronger version of Lemma 7.

▶ **Lemma 8.** *Suppose $\langle s,u\rangle\ [\varphi] \to_\mathcal{R} \langle t,v\rangle\ [\psi]$ at a position of the form $1p$.*
1. *If γ respects φ then $s\gamma \to_\mathcal{R} t\delta$ and $u\gamma = v\delta$ for some substitution δ respecting ψ.*
2. *If δ respects ψ then $s\gamma \to_\mathcal{R} t\delta$ and $u\gamma = v\delta$ for some substitution γ respecting φ.*

Proof. Note that whenever $\langle t_1,t_2\rangle \sim w$ for terms t_1 and t_2 then w must be of the form $\langle w_1,w_2\rangle$, by the definition of $\sim$ and because respectful substitutions introduce only values. We can therefore consider a step

$$\langle s,u\rangle\ [\varphi] \sim \langle s',u'\rangle\ [\varphi'] \xrightarrow[1p]{} \langle t',v'\rangle\ [\psi'] \sim \langle t,v\rangle\ [\psi] \tag{1}$$

1. Since γ respects φ there is some γ' respecting φ' such that $\langle s,u\rangle\gamma = \langle s',u'\rangle\gamma'$.
   First, if (1) involves a rule step then $\varphi' = \psi'$. Hence $\langle s',u'\rangle\gamma' \to_{\mathsf{rule},1p} \langle t',v'\rangle\gamma'$ by Lemma 6 and we have $u'\gamma' = v'\gamma'$. Because γ' respects ψ' and $\langle t',v'\rangle\ [\psi'] \sim \langle t,v\rangle\ [\psi]$ there is some δ respecting ψ such that $\langle t',v'\rangle\gamma' = \langle t,v\rangle\delta$. We thus have $s\gamma = s'\gamma' \to_\mathcal{R} t'\gamma' = t\delta$ and $u\gamma = u'\gamma' = v'\gamma' = v\delta$.
   Second, suppose (1) involves a calculation step. By the definition of $\to_{\mathsf{calc}}$ we have $\psi' = (\varphi' \wedge x = f(s_1,\ldots,s_n))$ for some $x \in \mathcal{V}$, $f \in \mathcal{F}_{\mathrm{theory}}$, and $s_1,\ldots,s_n \in \mathcal{V}\mathrm{ar}(\varphi')\cup\mathcal{V}\mathrm{al}$. So $f(s_1,\ldots,s_n)\gamma' \in \mathcal{T}(\mathcal{F}_{\mathrm{theory}})$ since γ' respects φ'. For w being the value symbol corresponding to $f(s_1,\ldots,s_n)\gamma'$, the substitution β given by $\beta(y) = w$ if $y = x$ and $\beta(y) = \gamma'(y)$ otherwise, respects ψ' and satisfies both $s'\gamma' \to_{\mathsf{calc}} t'\beta$ and $u'\gamma' = v'\beta$ because x is fresh. Because $\langle t',v'\rangle\ [\psi'] \sim \langle t,v\rangle\ [\psi]$ there is some δ respecting ψ such that $\langle t',v'\rangle\beta = \langle t,v\rangle\delta$. So $s\gamma = s'\gamma' \to_\mathcal{R} t'\beta = t\delta$ and $u\gamma = u'\gamma' = v'\beta = v\delta$.
2. Similar, see the appendix. ◀

We conclude this section with an auxiliary result relating rewrite steps on constrained term pairs to steps on unconstrained term pairs.

▶ **Lemma 9.** *Suppose the LCTRS $\mathcal{R}$ admits a rewrite step $\langle s,u\rangle\ [\varphi] \to_\mathcal{R} \langle t,u\rangle\ [\psi]$. For all substitutions γ and δ and all contexts C,*
1. *if $C[s\gamma] \xleftrightarrow[s\approx u\ [\varphi]]{} C[u\gamma]$ then $C[s\gamma] \xrightarrow{\mathcal{R}} \cdot \xleftrightarrow[t\approx u\ [\psi]]{} C[u\gamma]$, and*
2. *if $C[t\delta] \xleftrightarrow[t\approx u\ [\psi]]{} C[u\delta]$ then $C[t\delta] \xleftarrow{\mathcal{R}} \cdot \xleftrightarrow[s\approx u\ [\varphi]]{} C[u\delta]$.*

Proof.

1. The substitution γ respects φ. By Lemma 8(1) there is some δ respecting ψ such that $s\gamma \to_{\mathcal{R}} t\delta$ and $u\gamma = u\delta$. Hence $C[s\gamma] \xrightarrow[\mathcal{R}]{} C[t\delta] \xleftrightarrow[t \approx u\ [\psi]]{} C[u\delta] = C[u\gamma]$.

2. The substitution δ respects ψ. By Lemma 8(2) there is some γ respecting φ such that $s\gamma \to_{\mathcal{R}} t\delta$ and $u\gamma = u\delta$. Hence $C[t\delta] \xleftarrow[\mathcal{R}]{} C[s\gamma] \xleftrightarrow[s \approx u\ [\varphi]]{} C[u\gamma] = C[u\delta]$. ◄

3 Constrained Reduction Orders

To ensure termination of the resulting system, completion procedures rely on reduction orders. In [13] the following definition of a recursive path order was given.

▶ **Definition 10.** Suppose $\mathcal{F}_{\text{theory}}$ contains a symbol $>_\iota$ for every sort ι occurring in $\mathcal{F}_{\text{theory}}$ such that $>_\iota$ is interpreted as a (partial) well-founded order $\sqsupset_\iota$ on $\mathcal{I}_\iota$. Moreover, let $>^p$ be a precedence on $\mathcal{F}_{\text{terms}} \setminus \mathcal{F}_{\text{theory}}$. For terms s and t and constraint φ

1. $s \geqslant^{\mathsf{rpo}}_{[\varphi]} t$ if one of the following alternatives applies:
   a. $s, t \in \mathcal{T}(\mathcal{F}_{\text{theory}}, \mathcal{V}\mathsf{ar}(\varphi))$ and $\varphi \Rightarrow (s = t \vee s >_{\text{sort}(s)} t)$ is valid,
   b. $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$ with $f \notin \mathcal{F}_{\text{theory}}$ and $s_i \geqslant^{\mathsf{rpo}}_{[\varphi]} t_i$ for all $1 \leqslant i \leqslant n$,
   c. $s >^{\mathsf{rpo}}_{[\varphi]} t$, or $s = t$ and $s \in \mathcal{V}$;

2. $s >^{\mathsf{rpo}}_{[\varphi]} t$ if one of the following alternatives applies:
   a. $s, t \in \mathcal{T}(\mathcal{F}_{\text{theory}}, \mathcal{V}\mathsf{ar}(\varphi))$ and $\varphi \Rightarrow s >_{\text{sort}(s)} t$ is valid,
   b. $s = f(s_1, \ldots, s_n)$ for some $f \notin \mathcal{F}_{\text{theory}}$ and one of
      i. $s_i \geqslant^{\mathsf{rpo}}_{[\varphi]} t$ for some $1 \leqslant i \leqslant n$,
      ii. $t = g(t_1, \ldots, t_m)$, either $g \in \mathcal{F}_{\text{theory}}$ or $f >^p g$, and $s >^{\mathsf{rpo}}_{[\varphi]} t_j$ for all $1 \leqslant j \leqslant m$,
      iii. $t = f(t_1, \ldots, t_n)$, $s_i \geqslant^{\mathsf{rpo}}_{[\varphi]} t_i$ for all $1 \leqslant i \leqslant n$ and $s_i >^{\mathsf{rpo}}_{[\varphi]} t_i$ for some $1 \leqslant i \leqslant n$,
      iv. $t \in \mathcal{V}\mathsf{ar}(\varphi)$.

We now generalize this notion.

▶ **Definition 11.** A ternary relation $>_{[\cdot]}$ on $\mathcal{T}(\mathcal{F}, \mathcal{V}) \times \mathcal{T}_{\mathsf{bool}}(\mathcal{F}_{\text{theory}}, \mathcal{V}) \times \mathcal{T}(\mathcal{F}, \mathcal{V})$ is a *constrained reduction order* if there exists a reduction order $>$ such that $s >_{[\varphi]} t$ if and only if $s\gamma > t\gamma$ for all substitutions γ that respect φ.

The relation $>^{\mathsf{rpo}}_{[\cdot]}$ is a constrained reduction order according to this definition (Lemma 9 in the full version of [13]).

▶ **Example 12.**

1. Any reduction order $>$ gives rise to a constrained reduction order in which the constraints are simply ignored: setting $s >_{[\varphi]} t$ for all φ whenever $s > t$ vacuously satisfies the definition.

2. Let $n \sqsupset_{\mathsf{int}} m$ if $m \geqslant 0$ and $n > m$. For the LCTRS from Example 2, rule (1) can be oriented using the recursive path order by condition 2.b.ii in Definition 10. For rule (2) and $\varphi = x - 1 \geqslant 0$, we have $x >^{\mathsf{rpo}}_{[\varphi]} x - 1$ by condition 2.a since the implication $x - 1 \geqslant 0 \implies x > x - 1$ is valid. From this we obtain $\mathsf{fact}(x) >^{\mathsf{rpo}}_{[\varphi]} \mathsf{fact}(x - 1)$ by condition 2.b.iii. Moreover, $\mathsf{fact}(x) >^{\mathsf{rpo}}_{[\varphi]} x$ by 2.b.i such that $\mathsf{fact}(x) >^{\mathsf{rpo}}_{[\varphi]} \mathsf{fact}(x - 1) \cdot x$ follows from 2.b.ii. Note that the rules could not have been oriented by RPO when ignoring the constraints.

3. A well-founded $\mathcal{F}$-algebra $\mathcal{A}$ extending $\mathcal{I}$ with monotone (wrt $\sqsupseteq_{\mathsf{int}}$) interpretations for the function symbols in $\mathcal{F}_{\text{terms}}$ gives rise to a constrained reduction order: $s >_{[\varphi]} t$ if and only if $[\alpha]_{\mathcal{A}}(s) > [\alpha]_{\mathcal{A}}(t)$ is satisfied for all substitutions α that respect φ. Moreover, if the carrier of $\mathcal{A}$ is $\mathcal{V}\mathsf{al}$ then compatibility is decidable whenever the language of constraints is. For instance, consider again Example 2 and let $\mathcal{A}$ extend $\mathcal{I}$ by the monotone interpretation $\mathsf{fact}_{\mathcal{A}}(x) = (x+1)! + 1$ if $x \geqslant 0$ and $\mathsf{fact}_{\mathcal{A}}(x) = 2$ otherwise. We have

$$\mathsf{fact}_{\mathcal{A}}(x) = 2 > 1 = 1_{\mathcal{A}} \qquad\qquad\qquad \text{for all } x \leqslant 0$$
$$\mathsf{fact}_{\mathcal{A}}(x) = (x+1)! + 1 > (x! + 1) \cdot x = \mathsf{fact}_{\mathcal{A}}(x -_{\mathcal{A}} 1) \cdot_{\mathcal{A}} x \qquad\qquad \text{for all } x \geqslant 1$$

▶ **Lemma 13.** *Let $>_{[\cdot]}$ be a constrained reduction order and let φ and ψ be constraints.*
1. *The relation $>_{[\varphi]}$ is transitive.*
2. *If $s >_{[\varphi]} t$ and σ respects φ then $s\sigma >_{[\varphi]} t\sigma$.*
3. *If $s >_{[\varphi]} t$ then $C[s] >_{[\varphi]} C[t]$.*
4. *If $\psi \Rightarrow \varphi$ is valid and $\mathcal{V}\mathsf{ar}(\varphi) \subseteq \mathcal{V}\mathsf{ar}(\psi)$ then $s >_{[\varphi]} t$ implies $s >_{[\psi]} t$.*

▶ **Lemma 14.** *If $s\ [\psi] \to^{\sigma}_{\alpha,p} t\ [\psi]$ using $\alpha \colon \ell \to r\ [\varphi]$ satisfies $\ell >_{[\varphi]} r$ then $s >_{[\psi]} t$.*

Proof. By assumption $\psi \Rightarrow \varphi\sigma$ is valid and $\sigma(x) \in \mathcal{V}\mathsf{al} \cup \mathcal{V}\mathsf{ar}(\psi)$ for all $x \in \mathcal{V}\mathsf{ar}(\varphi)$. Now suppose γ is a substitution which respects ψ, i.e., the constraint $\psi\gamma$ is valid and $\gamma(x) \in \mathcal{V}\mathsf{al}$ for all $x \in \mathcal{V}\mathsf{ar}(\psi)$. Then $\varphi\sigma\gamma$ is valid and $(\sigma\gamma)(x) \in \mathcal{V}\mathsf{al}$ for all $x \in \mathcal{V}\mathsf{ar}(\varphi)$, so $\sigma\gamma$ respects φ. From $\ell >_{[\varphi]} r$ we thus obtain $s|_p\gamma = \ell\sigma\gamma > r\sigma\gamma = t|_p\gamma$, and hence $s\gamma > t\gamma$ by Lemma 13(3). ◀

A *reduction pair* $(>, \geqslant)$ consists of a reduction order $>$ and a reduction preorder $\geqslant$ such that $\to_{\mathsf{calc}} \subseteq \geqslant$ and $\geqslant \cdot > \cdot \geqslant\ \subseteq\ >$.

▶ **Lemma 15.** *If there is a reduction pair $(>, \geqslant)$ such that $\ell >_{[\varphi]} r$ for all $\ell \to r\ [\varphi] \in \mathcal{R}$ then $\mathcal{R}$ is terminating.*

Proof. By Lemma 14, the inclusion $\to_{\mathsf{calc}} \subseteq \geqslant$, and the compatibility of $>$ and $\geqslant$. ◀

Kop and Nishida (Lemmata 6 and 8 in the full version of [13]) showed that a suitable reduction preorder exists for the recursive path ordering. Similar to the case of plain term rewrite systems, any terminating LCTRS induces a constrained reduction order.

▶ **Lemma 16.** *If $\mathcal{R}$ is a terminating LCTRS then the relation defined by $s >_{[\varphi]} t$ if and only if $s\ [\varphi] \to^{+}_{\mathsf{rule}} t\ [\varphi]$ is a constrained reduction order.*

Proof. Let $>$ be the relation such that $s > t$ if and only if $s\ [\mathsf{true}] \to^{+}_{\mathsf{rule}} t\ [\mathsf{true}]$. Since $\mathcal{R}$ is terminating $>$ is a reduction order. So by Lemma 6 all substitutions γ respecting φ satisfy

$$s >_{[\varphi]} t \iff s\ [\varphi] \to^{+}_{\mathsf{rule}} t\ [\varphi] \implies s\gamma \to^{+}_{\mathsf{rule}} t\gamma \iff s\gamma > t\gamma \qquad\qquad ◀$$

The following example shows that a constrained reduction order is not necessarily compatible with the equivalence relation $\sim$ on constrained terms in the sense that $s\ [\varphi] \sim s'\ [\psi]$ and $s >_{[\varphi]} t$ imply $s' >_{[\psi]} t$.

▶ **Example 17.** For instance, for the constrained reduction order $>^{\mathsf{rpo}}_{[\cdot]}$ we have $\mathsf{f}(x) >_{[x=0]} x$ and $\mathsf{f}(x)\ [x = 0] \sim \mathsf{f}(0)\ [\mathsf{true}]$ but $\mathsf{f}(0) >_{[\mathsf{true}]} x$ does not hold.

We conclude this section by comparing our concept of a constrained reduction pair to definitions from the literature. Our notion resembles the definition by Falke and Kapur [5, Definition 23] for the theory of Peano arithmetic ($\mathcal{PA}$). Whereas they demand $C[s]$ ($\geqslant \cap \leqslant$) $C[t]$ for all $s \leftrightarrow^*_{\mathcal{PA}} t$, we use the more relaxed condition $\rightarrow_{\mathsf{calc}} \subseteq \geqslant$.

Fuhs et al. [6] define the order pair $(\succ_{\mathcal{P}ol}, \succeq_{\mathcal{P}ol})$ as a reduction pair processor for integer term rewrite systems. Since the order pair is based on *max-polynomial* interpretations, which are not strictly monotone, the resulting order on terms in not a reduction order, and hence does not produce a constrained reduction order in our sense.

4 Critical Pair Lemma

In this section we establish the Critical Pair Lemma, which is a key result to obtain confluence of the result of our completion procedure described in the next section.

▶ **Definition 18.** An *overlap* of an LCTRS $\mathcal{R}$ is a triple $\langle \ell_1 \rightarrow r_1 \ [\varphi_1], p, \ell_2 \rightarrow r_2 \ [\varphi_2] \rangle$ satisfying the following properties:
- $\ell_1 \rightarrow r_1 \ [\varphi_1]$ and $\ell_2 \rightarrow r_2 \ [\varphi_2]$ are variable-disjoint variants of rewrite rules in $\mathcal{R} \cup \mathcal{R}_{\mathsf{calc}}$,
- $p \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(\ell_2)$,
- ℓ_1 and $\ell_2|_p$ are unifiable with mgu σ and $\sigma(x) \in \mathcal{T}(\mathcal{F}_{\text{theory}}, \mathcal{V})$ for all $x \in \mathcal{V}\mathsf{ar}(\varphi_1) \cup \mathcal{V}\mathsf{ar}(\varphi_2)$,
- $\varphi_1\sigma \wedge \varphi_2\sigma$ is satisfiable, and
- if $p = \epsilon$ then $\ell_1 \rightarrow r_1 \ [\varphi_1]$ and $\ell_2 \rightarrow r_2 \ [\varphi_2]$ are not variants, or $\mathcal{V}\mathsf{ar}(r_1) \nsubseteq \mathcal{V}\mathsf{ar}(\ell_1)$.

In this case $\ell_2\sigma[r_1\sigma]_p \approx r_2\sigma \ [\varphi_1\sigma \wedge \varphi_2\sigma]$ is a *constrained critical pair*. The set of all constrained critical pairs of $\mathcal{R}$ is denoted by $\mathsf{CP}(\mathcal{R})$.

Note that in the last condition also $\mathcal{V}\mathsf{ar}(r_2) \nsubseteq \mathcal{V}\mathsf{ar}(\ell_2)$ since we may assume that the two rules are variants.

▶ **Example 19.** Consider the following rewrite rules:

$$(1) \quad \mathsf{f}(x) \rightarrow \mathsf{g}(x,x) \quad [x \leqslant 0] \qquad (3) \quad \mathsf{g}(\mathsf{f}(x), y) \rightarrow \mathsf{g}(x, z) \quad [x > 0 \wedge z > x]$$
$$(2) \quad \mathsf{h}(\mathsf{f}(x)) \rightarrow \mathsf{h}(x) \qquad [x \geqslant 0] \qquad (4) \quad \mathsf{g}(x, x + y) \rightarrow \mathsf{f}(y) \qquad [x > 0 \wedge y > 0]$$

The constrained critical pair $\mathsf{h}(\mathsf{g}(x,x)) \approx \mathsf{h}(x) \ [x \leqslant 0 \wedge x \geqslant 0]$ is obtained from the overlap $\langle (1), 1, (2) \rangle$. There is also an overlap $\langle (3), \epsilon, (3') \rangle$ between rule (3) and a renamed version $(3')$ of itself, which gives rise to the critical pair $\mathsf{g}(x, z) \approx \mathsf{g}(x, w) \ [x > 0 \wedge z > x \wedge w > x]$. Finally, the constrained critical pair $\mathsf{g}(x, z) \approx \mathsf{f}(y) \ [x > 0 \wedge y > 0 \wedge z = x + y]$ originates from the overlap $\langle \mathcal{R}_{\mathsf{calc}}, 2, (4) \rangle$. There is no constrained critical pair between rules (1) and (3) since the conjunction $x \leqslant 0 \wedge x > 0 \wedge z > x$ is not satisfiable. There is also no critical pair between (3) and (4) because any mgu σ of $\mathsf{g}(\mathsf{f}(x), y)$ and $\mathsf{g}(x', x' + y')$ assigns (a variant of) $\mathsf{f}(x)$ to x', violating the third condition in Definition 18.

▶ **Lemma 20** (Constrained Critical Pair Lemma). *Let $\mathcal{R}$ be an LCTRS. If $t \ {}_{\mathcal{R}}\!\leftarrow s \rightarrow_{\mathcal{R}} u$ then $t \downarrow_{\mathcal{R}} u$ or $t \leftrightarrow_{\mathsf{CP}(\mathcal{R})} u$.*

Proof. We abbreviate $\ell_1 \rightarrow r_1 \ [\varphi_1]$ by α_1 and $\ell_2 \rightarrow r_2 \ [\varphi_2]$ by α_2, and consider a peak

$$t \xleftarrow[\alpha_1]{p_1, \sigma_1} s \xrightarrow[\alpha_2]{p_2, \sigma_2} u$$

where σ_1 and σ_2 denote the employed substitutions. We distinguish three cases.

(1) If $p_1 \parallel p_2$ then $t \xrightarrow[\alpha_2]{p_2, \sigma_2} s \xleftarrow[\alpha_1]{p_1, \sigma_1} u$ since the same substitutions can be used for the respective steps such that constraints are still respected.

Otherwise one position must be above the other one. Without loss of generality we assume that $p_1 \leqslant p_2$, so there must be a position p such that $p_2 = p_1 p$. We further assume that the rewrite rules α_1 and α_2 have no variables in common. Hence $\mathcal{D}om(\sigma_1) \cap \mathcal{D}om(\sigma_2) = \varnothing$ and the substitution $\sigma = \sigma_1 \cup \sigma_2$ is well-defined. We distinguish two further cases depending on whether the peak is an instance of an overlap.

(2) Suppose $\langle \alpha_1, p, \alpha_2 \rangle$ is an overlap. Let γ be a most general unifier of $\ell_2|_p$ and ℓ_1. We have $\ell_2 \gamma [r_1 \gamma]_p \approx r_2 \gamma \; [\varphi_1 \gamma \wedge \varphi_2 \gamma] \in \mathsf{CP}(\mathcal{R})$. The substitution σ is a unifier of $\ell_2|_p$ and ℓ_1 because $(\ell_2|_p)\sigma = (\ell_2 \sigma_2)|_p = \ell_1 \sigma_1 = \ell_1 \sigma$. Consequently there exists a substitution τ such that $\sigma = \gamma \tau$. Since validity of $\varphi_1 \sigma = \varphi_1 \gamma \tau$ and $\varphi_2 \sigma = \varphi_2 \gamma \tau$ implies validity of $(\varphi_1 \gamma \wedge \varphi_2 \gamma)\tau$ we have

$$\ell_2 \sigma_2 [r_1 \sigma_1]_p = (\ell_2 \gamma [r_1 \gamma]_p)\tau \leftrightarrow_{\mathsf{CP}(\mathcal{R})} (r_2 \gamma)\tau = r_2 \sigma_2$$

and hence also $t \leftrightarrow_{\mathsf{CP}(\mathcal{R})} u$.

(3) Since $\varphi_1 \sigma_1 \wedge \varphi_2 \sigma_2 = \varphi_1 \sigma \wedge \varphi_2 \sigma = (\varphi_1 \gamma \wedge \varphi_2 \gamma)\tau$ is valid, the constraint $\varphi_1 \gamma \wedge \varphi_2 \gamma$ is satisfiable. So if $\langle \alpha_1, p, \alpha_2 \rangle$ is not an overlap then either $p = \epsilon$ and α_1 and α_2 are variants with $\mathcal{V}ar(r_1) \subseteq \mathcal{V}ar(\ell_1)$, or $p \notin \mathcal{P}os_{\mathcal{F}}(\ell_2)$. In the first case also $\mathcal{V}ar(r_2) \subseteq \mathcal{V}ar(\ell_2)$ must hold, which implies $r_1 \sigma_1 = r_2 \sigma_2$ and $t = u$. In the second case, there must be positions q_1 and q_2 such that $p = q_1 q_2$ and q_1 is a variable position in ℓ_2. Let x be the variable at $\ell_2|_{q_1}$, so $\sigma_2(x)|_{q_2} = \ell_1 \sigma_1$. We define the substitution σ_2' as

$$\sigma_2'(y) = \begin{cases} \sigma_2(y)[r_1 \sigma_1]_{q_2} & \text{if } y = x \\ \sigma_2(y) & \text{if } y \neq x \end{cases}$$

Clearly the step $\sigma_2(x) \rightarrow_{q_2, \alpha_1, \sigma_1} \sigma_2'(x)$ is valid. Hence $r_2 \sigma_2 \rightarrow^* r_2 \sigma_2'$ and $\ell_2 \sigma_2 [r_1 \sigma_1]_p = \ell_2 \sigma_2 [\sigma_2'(x)]_{q_1} \rightarrow^* \ell_2 \sigma_2'$. The substitution σ_2' also respects φ_2. This can be seen as follows. Since σ_2 respects φ_2 we have $\sigma_2(y) \in \mathcal{V}al$ for all $y \in \mathcal{V}ar(\varphi_2)$. So $x \notin \mathcal{V}ar(\varphi_2)$ as $x\sigma_2 \trianglerighteq \ell_1$ and left-hand sides of rules cannot be values. Therefore $\sigma_2(y) = \sigma_2'(y)$ holds for all $y \in \mathcal{V}ar(\alpha_2)$, and we have $\varphi_2 \sigma_2 = \varphi_2 \sigma_2'$. In summary, there exists a joining sequence

$$\ell_2 \sigma_2 [r_1 \sigma_1]_p \xrightarrow[\ell_1 \to r_1 \; [\varphi_1]]{\sigma_1} {}^* \ell_2 \sigma_2' \xrightarrow[\ell_2 \to r_2 \; [\varphi_2]]{\epsilon, \sigma_2'} r_2 \sigma_2' {}^* \xleftarrow[\ell_1 \to r_1 \; [\varphi_1]]{\sigma_1} r_2 \sigma_2 \qquad \blacktriangleleft$$

Extended Critical Pairs. For ordered rewriting in a constrained setting, we consider a reduction pair $(>, \geqslant)$ that gives rise to a constrained reduction order $>_{[\cdot]}$ and define

$$\mathcal{E}^> = \{ u\gamma \to v\gamma \; [\varphi\gamma] \mid u \approx v \; [\varphi] \in \mathcal{E}^{\pm} \text{ and } u\gamma >_{[\varphi\gamma]} v\gamma \}$$

for any set of constrained equations $\mathcal{E}$. Moreover, $\mathsf{CP}^>(\mathcal{E})$ denotes all constrained critical pairs originating from an overlap $\langle \ell_1 \approx r_1 \; [\varphi_1], p, \ell_2 \approx r_2 \; [\varphi_2] \rangle$ with most general unifier σ such that $\ell_1 \to r_1 \; [\varphi_1], \ell_2 \to r_2 \; [\varphi_2] \in \mathcal{E}^{\pm}$ and neither $r_1 \sigma >_{[\varphi_1 \sigma]} \ell_1 \sigma$ nor $r_2 \sigma >_{[\varphi_2 \sigma]} \ell_2 \sigma$. A reduction order is called *complete* for a set of constrained equations $\mathcal{E}$ if $s \leftrightarrow^*_{\mathcal{E}} t$ implies $s > t$, $s < t$, or $s = t$ for all ground terms s and t. The proof of the next result is in the appendix.

▶ **Lemma 21.** *If $>$ is complete for $\mathcal{R} \cup \mathcal{E}$ and $\mathcal{R} \subseteq >_{[\cdot]}$ then the inclusion $\xleftrightarrow[\mathsf{CP}(\mathcal{R} \cup \mathcal{E}^>)]{} \subseteq \xleftrightarrow[\mathsf{CP}^>(\mathcal{R} \cup \mathcal{E})]{} \cup \downarrow_{\mathcal{R} \cup \mathcal{E}^>}$ holds on ground terms.*

5 Abstract Completion

In this section we define an abstract inference system that can be instantiated to both standard and ordered completion. We consider a fixed reduction pair $(>, \geqslant)$ that gives rise to a constrained reduction order $>_{[\cdot]}$.

Table 1 The inference rules of CKB.

deduce
$$\frac{\mathcal{E}, \mathcal{R}}{\mathcal{E} \cup \{s \approx t \,[\varphi]\}, \mathcal{R}} \qquad \text{if } \langle s, u \rangle \,[\varphi] \,_{\mathcal{R} \cup \mathcal{E}^{\pm}} \leftarrow \langle u, u \rangle \,[\varphi] \rightarrow_{\mathcal{R} \cup \mathcal{E}^{\pm}} \langle u, t \rangle \,[\varphi]$$

compose
$$\frac{\mathcal{E}, \mathcal{R} \uplus \{s \rightarrow t \,[\varphi]\}}{\mathcal{E}, \mathcal{R} \cup \{s \rightarrow u \,[\psi]\}} \qquad \text{if } \langle s, t \rangle \,[\varphi] \rightarrow_{\mathcal{R} \cup \mathcal{E}^{>}} \langle s, u \rangle \,[\psi] \text{ and } s >_{[\varphi]} u$$

orient
$$\frac{\mathcal{E} \uplus \{s \simeq t \,[\varphi]\}, \mathcal{R}}{\mathcal{E}, \mathcal{R} \cup \{s \rightarrow t \,[\varphi]\}} \qquad \text{if } s >_{[\varphi]} t \text{ and } \mathsf{root}(s) \in \mathcal{F}_{\text{terms}} \setminus \mathcal{F}_{\text{theory}}$$

simplify
$$\frac{\mathcal{E} \uplus \{s \simeq t \,[\varphi]\}, \mathcal{R}}{\mathcal{E} \cup \{u \approx t \,[\psi]\}, \mathcal{R}} \qquad \text{if } \langle s, t \rangle \,[\varphi] \rightarrow_{\mathcal{R} \cup \mathcal{E}^{>}} \langle u, t \rangle \,[\psi]$$

delete
$$\frac{\mathcal{E} \uplus \{s \approx t \,[\varphi]\}, \mathcal{R}}{\mathcal{E}, \mathcal{R}} \qquad \text{if } s\gamma = t\gamma \text{ for all } \gamma \text{ respecting } \varphi$$

collapse
$$\frac{\mathcal{E}, \mathcal{R} \uplus \{t \rightarrow s \,[\varphi]\}}{\mathcal{E} \cup \{u \approx s \,[\psi]\}, \mathcal{R}} \qquad \text{if } \langle t, s \rangle \,[\varphi] \rightarrow_{\mathcal{R} \cup \mathcal{E}^{>}} \langle u, s \rangle \,[\psi]$$

$\mathsf{split}_{\mathcal{E}}$
$$\frac{\mathcal{E} \uplus \{s \simeq t \,[\varphi]\}, \mathcal{R}}{\mathcal{E} \cup \{s \approx t \,[\varphi \wedge \neg\psi], \ s \approx t \,[\varphi \wedge \psi]\}, \mathcal{R}} \qquad \text{if } \mathcal{V}\mathrm{ar}(\psi) \subseteq \mathcal{V}\mathrm{ar}(\varphi)$$

$\mathsf{split}_{\mathcal{R}}$
$$\frac{\mathcal{E}, \mathcal{R} \uplus \{s \rightarrow t \,[\varphi]\}}{\mathcal{E}, \mathcal{R} \cup \{s \rightarrow t \,[\varphi \wedge \neg\psi], \ s \rightarrow t \,[\varphi \wedge \psi]\}} \qquad \text{if } \mathcal{V}\mathrm{ar}(\psi) \subseteq \mathcal{V}\mathrm{ar}(\varphi)$$

▶ **Definition 22.** The inference system CKB of constrained (Knuth-Bendix) completion operates on pairs $(\mathcal{E}, \mathcal{R})$ consisting of constrained equations $\mathcal{E}$ and constrained rules $\mathcal{R}$, and consists of the inference rules of Table 1.

Below we provide some comments on the inference rules. The rewrite steps in e.g. simplify rewrite a *pair* $\langle s, t \rangle$ rather than a single term s. As observed in [7] it does not suffice to assume a rewrite step $s \,[\varphi] \rightarrow_{\mathcal{R}} u \,[\psi]$. For example, we have

$$\mathsf{f}(x + 0) \,[x > y] \sim \mathsf{f}(x + 0) \,[\mathsf{true}] \rightarrow_{\mathsf{calc}} \mathsf{f}(z) \,[z = x + 0] \sim \mathsf{f}(x) \,[\mathsf{true}] \sim \mathsf{f}(x) \,[x < y]$$

but it is not desirable to replace $y \approx \mathsf{f}(x + 0) \,[x > y]$ by $y \approx \mathsf{f}(x) \,[x < y]$. The same holds for compose, collapse, and deduce. The deduce rule is quite general in that it allows to add arbitrary equations that emerge from a peak between two $\mathcal{R} \cup \mathcal{E}$ steps. Below we will show that only (extended) critical pairs are necessary; hence as usual with completion procedures, an implementation will likely limit the application of deduce to these equations. Moreover, ordered rewriting is permitted in all rules that perform rewrite steps. Though this is uncommon in (unconstrained) standard completion, it gives more freedom to implementations and allows us to present only one set of inference rules for both settings. Furthermore, it is uncommon to perform a term comparison in compose. However, since additional variables may be introduced by $\rightarrow_{\mathsf{calc}}$ steps and $\sim$, $s >_{[\psi]} u$ need not hold. For instance, consider $>_{[\cdot]}$ defined as $s >_{[\cdot]} t$ if $s >^{\mathsf{lpo}} t$ holds, for some fixed reduction order $>^{\mathsf{lpo}}$ with precedence $\mathsf{f} > \mathsf{g}$. Given a rule $\mathsf{f}(x + y) \rightarrow \mathsf{g}(x + y) \,[\mathsf{true}]$, we have $\mathsf{f}(x + y) >_{[\mathsf{true}]} \mathsf{g}(x + y)$. We further have $\mathsf{g}(x + y) \rightarrow_{\mathsf{calc}} \mathsf{g}(z) \,[z = x + y]$ but $\mathsf{f}(x + y) >_{[z = x + y]} \mathsf{g}(z)$ does not hold.[1] Finally, the split rules are inspired by [8], they allow for a case distinction.

An inference step from equations and rules $(\mathcal{E}, \mathcal{R})$ to $(\mathcal{E}', \mathcal{R}')$ using one of the inference rules of Definition 22 is denoted by $(\mathcal{E}, \mathcal{R}) \vdash (\mathcal{E}', \mathcal{R}')$. We illustrate CKB on a concrete example, before presenting some basic properties related to inference steps.

---

[1] Note that if a pure $\rightarrow_{\mathsf{rule}}$ step is performed then $s >_{[\psi]} u$ is guaranteed by Lemma 14.

▶ **Example 23.** Consider the theory of integer arithmetic, RPO $>$ with precedence $\mathsf{h} > \mathsf{f} > \mathsf{g}$, and the following set of input equations:

(1) $\mathsf{f}(x,y) \approx \mathsf{f}(z,y) + 1 \ [x \geqslant 1 \wedge z = x - 1]$ (3) $\mathsf{f}(x,0) \approx \mathsf{g}(1,x) \ [x \leqslant 1]$

(2) $\mathsf{g}(0,y) \approx y \ [y \leqslant 0]$ (4) $\mathsf{h}(x) \approx \mathsf{f}(x,0) + 1$

We apply **orient** to equations (1) and (2) to obtain rewrite rules

(1) $\mathsf{f}(x,y) \rightarrow \mathsf{f}(z,y) + 1 \ [x \geqslant 1 \wedge z = x - 1]$ (2) $\mathsf{g}(0,y) \rightarrow y \ [y \leqslant 0]$

In rule (1) the variable z occurs on the right but not on the left-hand side, hence we **deduce** the critical pair

$$\mathsf{f}(w,y) + 1 \approx \mathsf{f}(z,y) + 1 \ [x \geqslant 1 \wedge z = x - 1 \wedge w = x - 1]$$

which can, however, be dropped using **delete**. (Note that a syntactic equality check of the equated terms would not suffice at this point.) Next we **orient** equation (3) from left to right. At this point the critical pair

$$\mathsf{g}(1,x) \approx \mathsf{f}(z,0) + 1 \ [x \geqslant 1 \wedge z = x - 1 \wedge x \leqslant 1]$$

results from the overlap $\langle (3), \epsilon, (1) \rangle$. But we can instead **deduce** the simpler, unconstrained equation (5) $\mathsf{g}(1,1) \approx \mathsf{f}(0,0) + 1$ as follows:

$$\mathsf{g}(1,1) \ [\mathsf{true}] \sim \mathsf{g}(1,x) \ [x = 1] \xleftarrow[\text{rule}]{(3)} \mathsf{f}(x,0) \ [x = 1] \sim \mathsf{f}(1,0) \ [\mathsf{true}] \sim \mathsf{f}(x,0) \ [x = 1 \wedge z = 0]$$

$$\xrightarrow[\text{rule}]{(1)} \mathsf{f}(z,0) + 1 \ [x = 1 \wedge z = 0] \sim \mathsf{f}(0,0) + 1 \ [\mathsf{true}]$$

When applying **simplify** with rule (3), equation (5) gets replaced by (6) $\mathsf{g}(1,1) \approx \mathsf{g}(1,0) + 1$. An application of **orient** produces the corresponding rule (6) $\mathsf{g}(1,1) \rightarrow \mathsf{g}(1,0) + 1$. A case split on $[x \leqslant 1]$ using $\mathsf{split}_\mathcal{E}$ followed by orientations replaces equation (4) by the rules (7) $\mathsf{h}(x) \rightarrow \mathsf{f}(x,0) + 1 \ [x \leqslant 1]$ and (8) $\mathsf{h}(x) \rightarrow \mathsf{f}(x,0) + 1 \ [\neg(x \leqslant 1)]$. Now **compose** using rule (3) can replace rule (7) by (9) $\mathsf{h}(x) \rightarrow \mathsf{g}(1,x) + 1 \ [x \leqslant 1]$. Since this is a pure rule step, $\mathsf{h}(x) >_{[x \leqslant 1]} \mathsf{g}(1,x) + 1$ holds by Lemma 14. Another application of **compose** applying rule (1) to rule (8) results in (10) $\mathsf{h}(x) \rightarrow \mathsf{f}(z,0) + 1 + 1 \ [x \geqslant 1 \wedge z = x - 1]$. We can apply **compose** again to (10), performing a calculation step to obtain (11) $\mathsf{h}(x) \rightarrow \mathsf{f}(x - 1, 0) + 2 \ [x \geqslant 1]$ (using $\mathsf{f}(z,y) + 2 \ [x \geqslant 1 \wedge z = x - 1] \sim \mathsf{f}(x - 1, y) + 2 \ [x \geqslant 1]$). Note that in both **compose** steps the orientation using $>_{[\cdot]}$ is preserved. At this point no equations are left, and all constrained critical pairs among the current set of rules

(1) $\mathsf{f}(x,y) \rightarrow \mathsf{f}(z,y) + 1 \ [x \geqslant 1 \wedge z = x - 1]$ (6) $\mathsf{g}(1,1) \rightarrow \mathsf{g}(1,0) + 1$

(2) $\mathsf{g}(0,y) \rightarrow y \ [x \leqslant 0]$ (9) $\mathsf{h}(x) \rightarrow \mathsf{g}(1,x) + 1 \ [x \leqslant 1]$

(3) $\mathsf{f}(x,0) \rightarrow \mathsf{g}(1,x) \ [x \leqslant 1]$ (11) $\mathsf{h}(x) \rightarrow \mathsf{f}(x - 1, 0) + 2 \ [x \geqslant 1]$

have been considered. Thus the system is complete according to Theorem 33 below.

▶ **Lemma 24.** *If* $(\mathcal{E}, \mathcal{R}) \vdash (\mathcal{E}', \mathcal{R}')$ *and the LCTRS* $\mathcal{R}$ *satisfies* $\mathcal{R} \subseteq >_{[\cdot]}$ *then also* $\mathcal{R}'$ *is an LCTRS such that* $\mathcal{R}' \subseteq >_{[\cdot]}$.

Proof. We show that any step $(\mathcal{E}, \mathcal{R}) \vdash (\mathcal{E}', \mathcal{R}')$ satisfies $\mathcal{R}' \setminus \mathcal{R} \subseteq >_{[\cdot]}$. If $(\mathcal{E}, \mathcal{R}) \vdash (\mathcal{E}', \mathcal{R}')$ applies **orient** or **compose** then this holds by definition. If $\mathsf{split}_\mathcal{R}$ was applied then $s >_{[\varphi \wedge \neg \psi]} t$ and $s >_{[\varphi \wedge \psi]} t$ follow from $s >_{[\varphi]} t$ by Lemma 13(4). The side condition $\mathsf{root}(s) \in \mathcal{F}_{\text{terms}} \setminus \mathcal{F}_{\text{theory}}$ in the **orient** rule ensures that $\mathcal{R}'$ is an LCTRS whenever $\mathcal{R}$ is.   ◀

We next show that the conversion relation associated with $(\mathcal{E}, \mathcal{R})$ is not changed by inference steps. Note that from now on we consider conversions between *unconstrained* terms, as opposed to the rewrite steps that are performed when applying inference rules.

▶ **Lemma 25.** *If* $(\mathcal{E}, \mathcal{R}) \vdash (\mathcal{E}', \mathcal{R}')$ *then* $\xrightarrow[\mathcal{E} \cup \mathcal{R}]{} \setminus \xleftrightarrow[\mathcal{E}' \cup \mathcal{R}']{} \subseteq \xrightarrow[\mathcal{R}' \cup \mathcal{E}'{>}]{=} \cdot \xleftrightarrow[\mathcal{E}']{=} \cdot \xleftarrow[\mathcal{R}' \cup \mathcal{E}'{>}]{=}.$

Proof. We perform a case distinction on the applied inference rule.

- If compose was applied then $\langle s, t \rangle \, [\varphi] \to_{\mathcal{R} \cup \mathcal{E}{>}} \langle s, u \rangle \, [\psi]$ implies $\langle t, s \rangle \, [\varphi] \to_{\mathcal{R} \cup \mathcal{E}{>}} \langle u, s \rangle \, [\psi]$. If there is a step $C[s\sigma] \to C[t\sigma]$ using $s \to t \, [\varphi]$ then σ respects φ. Applying Lemma 9(1) to $C[t\sigma] \leftarrow C[s\sigma]$ yields $C[t\sigma] \to_{\mathcal{R}} \cdot \leftrightarrow_{u \approx s \, [\psi]} C[s\sigma]$ and thus $C[s\sigma] \leftrightarrow_{s \approx u \, [\psi]} \cdot _{\mathcal{R}}{\leftarrow} C[t\sigma]$. As $s \to u \, [\psi] \in \mathcal{R}'$ and $\mathcal{R} \subseteq \mathcal{R}'$ we have $C[s\sigma] \to_{\mathcal{R}'} \cdot _{\mathcal{R}'}{\leftarrow} C[t\sigma]$.

- In the case orient was applied to $s \simeq t \, [\varphi] \in \mathcal{E}$ then for a step $C[s\sigma] \leftrightarrow C[t\sigma]$ we have $C[s\sigma] \to_{s \to t \, [\varphi]} C[t\sigma]$ or $C[t\sigma] \, _{s \to t \, [\varphi]}{\leftarrow} C[s\sigma]$. The claim holds because $s \to t \, [\varphi] \in \mathcal{R}'$.

- Suppose simplify was applied with $\langle s, t \rangle \, [\varphi] \to_{\mathcal{R} \cup \mathcal{E}{>}} \langle u, t \rangle \, [\psi]$. Given a step $C[s\sigma] \leftrightarrow C[t\sigma]$ using $s \approx t \, [\varphi]$, σ respects φ. By Lemma 9(1) we have $C[s\sigma] \to_{\mathcal{R}} \cdot \leftrightarrow_{u \approx t \, [\psi]} C[t\sigma]$. The claim holds because $u \approx t \, [\psi] \in \mathcal{E}'$.

- If delete was applied and there is a step $C[s\sigma] \leftrightarrow C[t\sigma]$ using the deleted equation $s \approx t \, [\varphi]$, then $\varphi\sigma$ is valid, so $s\sigma = t\sigma$ must hold.

- Suppose collapse was applied with $\langle t, s \rangle \, [\varphi] \to_{\mathcal{R} \cup \mathcal{E}{>}} \langle u, s \rangle \, [\psi]$. If $C[t\sigma] \leftrightarrow C[s\sigma]$ then $C[t\sigma] \to_{\mathcal{R}} \cdot \leftrightarrow_{u \approx t \, [\psi]} C[s\sigma]$ by Lemma 9(1). As $u \approx s \, [\psi] \in \mathcal{E}'$ and $\mathcal{R} \setminus \{t \to s \, [\varphi]\} \subseteq \mathcal{R}'$ we have $C[s\sigma] \leftrightarrow_{\mathcal{E}'} \cdot _{\mathcal{R}'}{\leftarrow} C[t\sigma]$.

- In the case of $\mathsf{split}_{\mathcal{E}}$ and a step $C[s\sigma] \leftrightarrow C[t\sigma]$ using $s \approx t \, [\varphi]$, the substitution σ respects φ. Since $\mathcal{V}\mathrm{ar}(\psi) \subseteq \mathcal{V}\mathrm{ar}(\varphi)$, σ must respect either ψ or $\neg\psi$.

- Similarly, in the case of $\mathsf{split}_{\mathcal{R}}$ and a step $C[s\sigma] \to C[t\sigma]$ using $s \to t \, [\varphi]$, the substitution σ respects φ. Since $\mathcal{V}\mathrm{ar}(\psi) \subseteq \mathcal{V}\mathrm{ar}(\varphi)$, σ must respect either ψ or $\neg\psi$ such that one of the two new rules can be applied. ◀

▶ **Lemma 26.** *If* $(\mathcal{E}, \mathcal{R}) \vdash (\mathcal{E}', \mathcal{R}')$ *then* $\xleftrightarrow[\mathcal{E}' \cup \mathcal{R}']{} \subseteq \xleftrightarrow[\mathcal{E} \cup \mathcal{R}]{*}.$

The proof of Lemma 26 is similar to that of Lemma 25 and can be found in the appendix. In combination the last two lemmata show the invariance of the conversion relation.

▶ **Corollary 27.** *If* $(\mathcal{E}, \mathcal{R}) \vdash^* (\mathcal{E}', \mathcal{R}')$ *then the relations* $\xleftrightarrow[\mathcal{E} \cup \mathcal{R}]{*}$ *and* $\xleftrightarrow[\mathcal{E}' \cup \mathcal{R}']{*}$ *coincide.*

We now follow the proof approach in [9] by showing that whenever a term multiset dominates a conversion in $(\mathcal{E}, \mathcal{R})$ then this property is preserved by CKB steps. To this end, we compare terms with the (well-founded) relation $\succ$ defined as the lexicographic combination of $>$ and $\to_{\mathsf{calc}}^+$, using $\geqslant$ as preorder. The relation $\succeq$ denotes its reflexive closure. We label rewrite steps by multisets of terms and write $s \xrightarrow[\mathcal{R}]{S} t$ if $s \to_{\mathcal{R}} t$ and there exist terms $s', t' \in S$ such that $s' \succeq s$ and $t' \succeq t$.

▶ **Lemma 28.** *If* $\mathcal{R} \subseteq >_{[\cdot]}$ *then* $\to_{\mathcal{R}} \subseteq \succ$.

Proof. For a rule step $s \to_{\mathsf{rule}, \mathcal{R}} t$ we have $s > t$ by Lemma 14 and hence $s \succ t$. A step $s \to_{\mathsf{calc}} t$ satisfies $s \geqslant t$ by assumption and $s \succ t$ by the definition of $\succ$. ◀

▶ **Lemma 29.** *If* $(\mathcal{E}, \mathcal{R}) \vdash (\mathcal{E}', \mathcal{R}')$ *and* $\mathcal{R} \subseteq >_{[\cdot]}$ *then* $\xleftrightarrow[\mathcal{E} \cup \mathcal{R}]{S}{}^* \subseteq \xleftrightarrow[\mathcal{E}' \cup \mathcal{R}']{S}{}^*.$

Proof. We consider a single step

$$C[t\sigma] \xleftarrow[t \approx u \, [\varphi]]{S} C[u\sigma]$$

such that $t \approx u \ [\varphi] \in \mathcal{E} \cup \mathcal{R}$ and σ respects φ, and show $C[t\sigma] \xleftrightarrow[\mathcal{E}' \cup \mathcal{R}']{S}{}^* C[u\sigma]$. The statement of the lemma follows then by induction on the length of the conversion. According to Lemma 25 there exist terms v and w that satisfy

$$C[t\sigma] \xrightarrow[\mathcal{R}']{=} v \xrightarrow[\mathcal{E}' \cup \mathcal{R}']{=} w \xleftarrow[\mathcal{R}']{=} C[u\sigma]$$

There must be terms t' and u' in S such that $t' \succeq C[t\sigma]$ and $u' \succeq C[u\sigma]$. From the assumption $\mathcal{R} \subseteq >_{[\cdot]}$ we obtain $C[t\sigma] \succeq v$ and $C[u\sigma] \succeq w$ by Lemmata 24 and 28 and thus $t' \succeq v$ and $u' \succeq w$. Hence all (non-empty) steps between $C[t\sigma]$ and $C[u\sigma]$ can be labeled by S such that $C[t\sigma] \xleftrightarrow[\mathcal{E}' \cup \mathcal{R}']{S}{}^* C[u\sigma]$. ◀

We now consider a *run*, that is, a finite sequence of the form

$$\Gamma \colon (\mathcal{E}_0, \mathcal{R}_0) \vdash (\mathcal{E}_1, \mathcal{R}_1) \vdash (\mathcal{E}_2, \mathcal{R}_2) \vdash \cdots \vdash (\mathcal{E}_n, \mathcal{R}_n)$$

where $\mathcal{R}_0 \subseteq >_{[\cdot]}$ is assumed.[2] Simple induction proofs using Lemma 24, Corollary 27, and Lemma 29 extend the respective results to the final system $\mathcal{E}_n \cup \mathcal{R}_n$ of the run.

▶ **Corollary 30.** *The inclusion $\mathcal{R}_n \subseteq >_{[\cdot]}$ holds.*

▶ **Corollary 31.** *The relations $\xleftrightarrow[\mathcal{E}_0 \cup \mathcal{R}_0]{}{}^*$ and $\xleftrightarrow[\mathcal{E}_n \cup \mathcal{R}_n]{}{}^*$ coincide.*

▶ **Corollary 32.** *The inclusion $\xleftrightarrow[\mathcal{E}_i \cup \mathcal{R}_i]{S}{}^* \subseteq \xleftrightarrow[\mathcal{E}_n \cup \mathcal{R}_n]{S}{}^*$ holds.*

5.1 Standard Completion

The run Γ is *successful* if $\mathcal{E}_n = \varnothing$ and the inclusion $\mathsf{CP}(\mathcal{R}_n) \subseteq \bigcup_{i=0}^{n} \xleftrightarrow[\mathcal{E}_i]{}$ holds.

▶ **Theorem 33.** *If Γ is successful then $\mathcal{R}_n$ is a complete presentation of $\mathcal{E}_0 \cup \mathcal{R}_0$.*

Proof. From Corollary 30 we obtain $\mathcal{R}_n \subseteq >_{[\cdot]}$ and thus $\mathcal{R}_n$ is terminating by Lemma 15. In order to establish confluence, consider a peak

$$t \xleftarrow[\mathcal{R}_n]{S_1} s \xrightarrow[\mathcal{R}_n]{S_2} u$$

From $\mathcal{R}_n \subseteq >_{[\cdot]}$ and Lemma 28 we obtain $s \succ t$ and $s \succ u$. Using Lemma 20 and the definition of success, two cases are distinguished.

- If $t \downarrow_{\mathcal{R}_n} u$ then all steps in this joining sequence can be labeled with $\{t, u\}$, again using Lemma 28.
- Suppose $t \leftrightarrow_{\mathcal{E}_i} u$ for some $i \geqslant 0$. We can label this step with $\{t, u\}$ and thus obtain $t \xleftrightarrow[\mathcal{R}_n]{\{t,u\}}{}^* u$ from Corollary 32 since $\mathcal{E}_n$ is empty.

In both cases there is a conversion $t \xleftrightarrow[\mathcal{R}_n]{\{t,u\}}{}^* u$. Since $s \succ t$ and $s \succ u$ imply $S_1 \succ_{\mathsf{mul}} \{t, u\}$ and $S_2 \succ_{\mathsf{mul}} \{t, u\}$, $\mathcal{R}_n$ is peak decreasing and hence confluent by Lemma 3. ◀

---

[2] Rather than requiring the usual $\mathcal{R}_0 = \varnothing$, the more general condition $\mathcal{R}_0 \subseteq >_{[\cdot]}$ is useful when the orientation of the input equations needs to be preserved.

5.2 Ordered Completion

In this section we assume that the reduction order $>$ is complete for $\mathcal{E}_0 \cup \mathcal{R}_0$, so by Corollary 31 complete for $\mathcal{E}_n \cup \mathcal{R}_n$. We use the following modified notion of success: Γ is successful if

$$\mathsf{CP}^>(\mathcal{R}_n \cup \mathcal{E}_n) \subseteq \bigcup_{i=0}^{n} \underset{\mathcal{E}_i}{\overset{}{\longleftrightarrow}}$$

holds. We write $\mathcal{S}_n$ for the LCTRS $\mathcal{R}_n \cup \mathcal{E}_n^>$.

▶ **Theorem 34.** *If Γ is successful then $\mathcal{S}_n$ is a ground complete presentation of $\mathcal{E}_0 \cup \mathcal{R}_0$.*

Proof. The LCTRS $\mathcal{S}_n$ is contained in $>_{[\cdot]}$ by Corollary 30 and the definition of $\mathcal{E}_n^>$, hence $\mathcal{S}_n$ is terminating by Lemma 15. For showing ground confluence we consider a ground peak

$$t \underset{\mathcal{S}_n}{\overset{S_1}{\longleftarrow}} s \underset{\mathcal{S}_n}{\overset{S_2}{\longrightarrow}} u$$

From the inclusion $\mathcal{S}_n \subseteq >_{[\cdot]}$ and Lemma 28 we obtain both $s \succ t$ and $s \succ u$. By Lemma 20 $t \downarrow_{\mathcal{S}_n} u$ or $t \leftrightarrow_{\mathsf{CP}(\mathcal{S}_n)} u$ hold. The latter in turn implies $t \downarrow_{\mathcal{S}_n} u$ or $t \leftrightarrow_{\mathsf{CP}^>(\mathcal{R}_n \cup \mathcal{E}_n)} u$ by Lemma 21. Taking the definition of success into account there are two possibilities.

- If $t \downarrow_{\mathcal{S}_n} u$ then all steps in this joining sequence can be labeled by $\{t, u\}$, using Lemma 28.
- If $t \leftrightarrow_{\mathcal{E}_i} u$ for some $i \geqslant 0$ then this step can be labeled with $\{t, u\}$ and therefore $t \underset{\mathcal{R}_n \cup \mathcal{E}_n}{\overset{\{t,u\}}{\longleftrightarrow}}{}^* u$ is obtained from Corollary 32. Then there also exists such a conversion between t and u where all intermediate terms are ground. Since the reduction order $>$ is assumed to be complete, $v \to_{\mathcal{S}_n} w$ or $w \mathrel{{}_{\mathcal{S}_n}{\leftarrow}} v$ for every step $v \leftrightarrow_{\mathcal{E}_n} w$ in this conversion. Hence $t \underset{\mathcal{S}_n}{\overset{\{t,u\}}{\longleftrightarrow}}{}^* u$ follows.

So in both cases we obtain a conversion $t \underset{\mathcal{S}_n}{\overset{\{t,u\}}{\longleftrightarrow}}{}^* u$. From $s \succ t$ and $s \succ u$ we obtain $S_1 \succ_{\mathsf{mul}} \{t, u\}$ and $S_2 \succ_{\mathsf{mul}} \{t, u\}$. Hence $\mathcal{S}_n$ is peak decreasing on ground terms with respect to $\succ_{\mathsf{mul}}$ and therefore ground confluent by Lemma 3. ◀

6 Implementation and Applications

We implemented the inference system CKB presented in Section 5 on top of the Ctrl tool [14] which now supports both standard Knuth-Bendix completion and ordered completion for LCTRSs. To establish termination of the resulting system, either RPO (Definition 10) with a user-specified precedence or the termination proving facilities already present in Ctrl can be used (which are rather powerful due to a DP framework for LCTRSs [12]). For the latter mode, we adapted the approach of [19] to the LCTRS setting. If desirable, the orientation of the input equations can be preserved (provided that the termination proving capabilities of Ctrl suffice, obviously). In the Ctrl infrastructure, the underlying theory can be specified by the user in a theory specification file. For common theories such as integers, bit vectors, strings, and matrices these specification files are already present. As SMT solvers we used Ctrl's internal solver and Z3 [4].

▶ **Example 35.** As one of many optimizations on the intermediate representation, LLVM provides the Instcombine pass to simplify expressions, comprising over 1000 simplification rules. In [15,16] about 500 of these rules were expressed in the domain-specific Alive language, which closely resembles constrained rewrite systems. We transformed this rule set into an

LCTRS, resulting in rules of the following shape:[3]

$$\mathsf{add}(x, x) \to \mathsf{shift\_left}(x, \#\mathsf{x}01) \tag{1}$$

$$\mathsf{add}(\mathsf{add}(\mathsf{xor}(\mathsf{or}(x, c_1), y), \#\mathsf{x}01), w) \to \mathsf{sub}(w, \mathsf{and}(x, c_2)) \; [c_1 =\sim c_2] \tag{2}$$

$$\mathsf{add}(\mathsf{xor}(x, c), z) \to \mathsf{sub}(c + z, x) \; [\mathsf{isPowerOf2}(c + \#\mathsf{x}01) \wedge \ldots] \tag{3}$$

Here the set of values comprises all bit vectors of a fixed length (e.g., 8). The set of function symbols $\mathcal{F}_{\text{theory}}$ adds logical, arithmetic, and shift operations on bit vectors, which allow to express auxiliary predicates like isPowerOf2. On the other hand $\mathcal{F}_{\text{terms}}$ consists of symbols such as add, sub, and xor which refer to the bit vector operations in programs that get replaced in the Instcombine pass. For example, rule (1) replaces addition of two equal numbers by a left-shift by one bit. Rule (2) simplifies two consecutive additions with bitwise operations in their arguments to a subtraction, but is only applicable if the constant c_1 is the bitwise negation of the constant c_2. Also rule (3) implements some bit twiddling, checking whether a constant is a power of 2 (among other constraints). Since the optimization set is community maintained and constantly in flux, unintended interaction and overlapping patterns are not uncommon, despite the expectation of the community that there be a "canonical" form for every expression. For example rule (2) admits a critical pair with rule (1).

Completing the entire system is beyond the reach for our tool, but Ctrl completes for instance 11 optimizations for expressions rooted by an addition to a system of 15 rules, thereby eliminating some sources of nonconfluence. We maintained the orientation of the input rules, and could use the termination prover present in Ctrl.

The next example illustrates that LCTRS completion can prevail over completion of standard rewrite systems even in the absence of constraints: using a "background theory" such as integer arithmetic may admit a much more succinct presentation.

▶ **Example 36.** The tool AQL[4] performs data integration, i.e., the transformation of data from one database scheme to another, by means of a category-theoretic approach which takes advantage of (ordered) completion [18]. More precisely, it attempts to get a complete presentation for a set of ground equations describing data in the first database, plus non-ground equations describing the transformation to the second schema. A (ground) complete system can then be used to build the initial term model describing the data in the second schema, by enumerating terms and rewriting them to normal form until a fixed point is reached. The following example is taken from AQL's problem suite:

$\mathsf{workAt}(\mathsf{mng}(e)) \approx \mathsf{workAt}(e)$	$\mathsf{workAt}(\mathsf{sec}(x)) \approx \mathsf{dep}(x)$	$\mathsf{age}_c(e) \approx \mathsf{age}(e) + \mathsf{age}(\mathsf{mng}(e))$
$\mathsf{first}(\mathsf{a}) \approx \texttt{"Alice"}$	$\mathsf{first}(\mathsf{b}) \approx \texttt{"Bob"}$	$\mathsf{first}(\mathsf{c}) \approx \texttt{"Carl"}$
$\mathsf{mng}(\mathsf{a}) \approx \mathsf{b}$	$\mathsf{mng}(\mathsf{b}) \approx \mathsf{b}$	$\mathsf{sec}(\mathsf{b}) \approx \mathsf{a}$
$\mathsf{workAt}(\mathsf{a}) \approx \mathsf{m}$	$\mathsf{dname}(\mathsf{s}) \approx \texttt{"CS"}$	$\mathsf{dname}(\mathsf{m}) \approx \texttt{"Math"}$

Here the set of values consists of integers and strings, and $\mathcal{F}_{\text{theory}}$ contains integer arithmetic. The signature $\mathcal{F}_{\text{terms}}$ contains symbols dep, workAt, mng, and sec to represent a database schema that describes departments and employees working therein, with relations to designate managers and secretaries. There are equations defining general relations between relations as in the first row, and many ground equations describing the actual data (the constants a,

---

[3] Full details can be found at `http://cl-informatik.uibk.ac.at/users/swinkler/lctrs_completion/`.
[4] `http://categoricaldata.net/aql.html`

b, c refer to entries in a database table). Not all equations of the latter type are shown due to reasons of space. Ctrl can easily complete this system of 22 equations to an LCTRS of 25 rules within less than a second. Input problems for AQL often relate to standard data types like integers and strings, thus it is a key advantage if, e.g., numbers and arithmetic are already present in the theory and do not need to be axiomatized explicitly.

We conclude this section with an example on ordered completion.

▶ **Example 37.** In its ordered completion mode, Ctrl can verify ground completeness of the following system describing sorting the elements in an unordered tree:

$$[\,] \,@\, xs \to xs \qquad\qquad (x : xs) \,@\, ys \to x : (xs \,@\, ys)$$
$$\mathsf{add}(x, [\,]) \to [x] \qquad\qquad \mathsf{add}(x, y : ys) \to x : (y : ys) \; [x < y]$$
$$\mathsf{add}(x, y : ys) \to y : \mathsf{add}(x, ys)) \; [x \geqslant y] \qquad\qquad \mathsf{sort}([\,]) \to [\,]$$
$$\mathsf{sort}(x : xs) \to \mathsf{add}(x, \mathsf{sort}(xs)) \qquad\qquad \mathsf{flatten}(\mathsf{L}(x)) \to [x]$$
$$\mathsf{flatten}(\mathsf{N}(x, y)) \to \mathsf{flatten}(x) \,@\, \mathsf{flatten}(y)$$
$$\mathsf{tsort}(t) \to \mathsf{sort}(\mathsf{flatten}(t)) \qquad\qquad \mathsf{N}(x, y) = \mathsf{N}(y, x)$$

where the logical constraints are expressed over the theory of integer arithmetic. Obviously, standard completion fails on this example because of the commutativity equation. For this example we used constrained RPO such that the resulting system is complete with respect to a ground-total reduction order.

7 Conclusion

In this paper we presented an abstract completion inference system for both standard and ordered completion of LCTRSs. We provide a new and succinct correctness proof. Our prototype implementation shows the potential of completion for this powerful rewriting concept also on practical examples.

A completion procedure for a different kind of constrained rewrite systems was already proposed in [8]. However, the work presented in this paper differs from this older approach in several crucial aspects. It is known [13] that LCTRSs can express systems where the version of constrained systems from [8] admits no finite presentation. Moreover, our inference system covers not only standard but also ordered completion, and we give full and novel proofs which are very concise due to the use of peak decreasingness. We also employ constrained instead of standard reduction orders, which gives a lot more flexibility to implementations and allows in particular to perform completion with termination tools. Finally, the implementation mentioned in [8] was restricted to integers and is no longer available.

For future research a variety of directions is conceivable. There are several opportunities to enhance efficiency and effectiveness of the tool, such as stronger termination techniques and critical pair criteria [2]. Theoretical results on infinite runs can shed light on the interesting case of systems generated in the limit. Finally, formalization of LCTRSs and respective completion procedures in a proof assistant would enhance reliability.

References

1 F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998. `doi:10.1017/CBO9781139172752`.

2 L. Bachmair. *Canonical Equational Proofs.* Birkhäuser, 1991.

3 L. Bachmair, N. Dershowitz, and D.A. Plaisted. Completion without failure. In H. Aït Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2: Rewriting Techniques of *Progress in Theoretical Computer Science*, pages 1–30. Academic Press, 1989.

4 L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. 14th TACAS*, volume 4963 of *LNCS*, pages 337–340, 2008. `doi:10.1007/978-3-540-78800-3_24`.

5 S. Falke and D. Kapur. Dependency pairs for rewriting with built-in numbers and semantic data structures. In *Proc. 19th RTA*, volume 5117 of *LNCS*, pages 94–109, 2008. `doi:10.1007/978-3-540-70590-1_7`.

6 C. Fuhs, J. Giesl, M. Plücker, P. Schneider-Kamp, and S. Falke. Proving termination of integer term rewriting. In *Proc. 20th RTA*, volume 5595 of *LNCS*, pages 32–47, 2009. `doi:10.1007/978-3-642-02348-4_3`.

7 C. Fuhs, C. Kop, and N. Nishida. Verifying procedural programs via constrained rewriting induction. *ACM TOCL*, 18(2):14:1–14:50, 2017. `doi:10.1145/3060143`.

8 Y. Furuichi, N. Nishida, M. Sakai, K. Kusakari, and T. Sakabe. Approach to procedural-program verification based on implicit induction of constrained term rewriting systems. *IPSJ Transactions on Programming*, 1(2):100–121, 2008. In Japanese.

9 N. Hirokawa, A. Middeldorp, and C. Sternagel. A new and formalized proof of abstract completion. In *Proc. 5th ITP*, volume 8558 of *LNCS*, pages 292–307, 2014. `doi:10.1007/978-3-319-08970-6_19`.

10 K. Hoder, Z. Khasidashvili, K. Korovin, and A. Voronkov. Preprocessing techniques for first-order clausification. In *Proc. 12th FMCAD*, pages 44–51, 2012.

11 D.E. Knuth and P. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970. `doi:10.1016/B978-0-08-012975-4`.

12 C. Kop. Termination of LCTRSs. In *Proc. 13th WST*, pages 59–63, 2013.

13 C. Kop and N. Nishida. Term rewriting with logical constraints. In *Proc. 9th FroCoS*, volume 8152 of *LNAI*, pages 343–358, 2013. Full version available at `https://www.cs.ru.nl/~cynthiakop/frocos13.pdf`. `doi:10.1007/978-3-642-40885-4_24`.

14 C. Kop and N. Nishida. Constrained Term Rewriting tooL. In *Proc. 20th LPAR*, volume 9450 of *LNAI*, pages 549–557, 2015. `doi:10.1007/978-3-662-48899-7_38`.

15 N. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Provably correct peephole optimizations with Alive. In *Proc. 36th PLDI*, pages 22–32, 2015. `doi:10.1145/2737924.2737965`.

16 N. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Practical verification of peephole optimizations with Alive. *Communications of the ACM*, 61(2):84–91, 2018. `doi:10.1145/3166064`.

17 A. Nadel. Bit-vector rewriting with automatic rule generation. In *Proc. 16th CAV*, pages 663–679, 2014. `doi:10.1007/978-3-319-08867-9_44`.

18 P. Schultz and R. Wisnesky. Algebraic data integration. *JFP*, 27(e24):51 pages, 2017. `doi:10.1017/S0956796817000168`.

19 I. Wehrman, A. Stump, and E.M. Westbrook. Slothrop: Knuth-Bendix completion with a modern termination checker. In *Proc. 17th RTA*, volume 4098 of *LNCS*, pages 287–296, 2006. `doi:10.1007/11805618_22`.

A Proofs

The following fact becomes useful in the sequel.

▶ **Lemma 38.** *If $\varphi \Rightarrow \psi$ is valid, $\mathcal{V}\mathsf{ar}(\psi) \subseteq \mathcal{V}\mathsf{ar}(\varphi)$, and γ respects φ then γ respects ψ.*

Proof of Lemma 8(2). Since δ respects ψ there is some δ' respecting ψ' such that $\langle t, v \rangle \delta = \langle t', v' \rangle \delta'$.

First, (1) involves a rule step then $\varphi' = \psi'$, $\langle s', u' \rangle \delta' \rightarrow_{\mathsf{rule}, 1p} \langle t', v' \rangle \delta'$, and $u'\delta' = v'\delta'$ by Lemma 6. Because $\langle s, u \rangle \, [\varphi] \sim \langle s', u' \rangle \, [\varphi']$ there is some γ such that $\langle s, u \rangle \gamma = \langle s', u' \rangle \delta'$. We thus have $s\gamma = s'\delta' \rightarrow_{\mathcal{R}} t'\delta' = t\delta$ and $u\gamma = u'\delta' = v'\delta' = v\delta$.

Next suppose (1) involves a calculation step. By the definition of $\rightarrow_{\mathsf{calc}}$ we have $\psi' = (\varphi' \wedge x = f(s_1, \ldots, s_n))$ for some $x \in \mathcal{V}$, $f \in \mathcal{F}_{\mathsf{theory}}$, and $s_1, \ldots, s_n \in \mathcal{V}\mathsf{ar}(\varphi') \cup \mathcal{V}\mathsf{al}$. Since x is fresh we have $s'\delta' \rightarrow_{\mathsf{calc}} t'\delta'$ and $u'\delta' = v'\delta'$. From $\langle s, u \rangle \, [\varphi] \sim \langle s', u' \rangle \, [\varphi']$ we obtain a substitution γ respecting φ such that $\langle s, u \rangle \gamma = \langle s', u' \rangle \delta'$. Therefore $s\gamma = s'\delta' \rightarrow_{\mathcal{R}} t'\delta' = t\delta$ and $u\gamma = u'\delta' = v'\delta' = v\delta$. ◀

Proof of Lemma 13.
1. For any γ that respects φ we have $s\gamma > t\gamma$ and $t\gamma > u\gamma$, hence $s\gamma > u\gamma$ follows from transitivity of $>$.
2. Suppose γ satisfies φ. We have to show that $(s\sigma)\gamma > (t\sigma)\gamma$ holds. By assumption σ respects φ, hence so does $\sigma\gamma$. From $s >_{[\varphi]} t$ we therefore obtain $s(\sigma\gamma) > t(\sigma\gamma)$, so also $(s\sigma)\gamma > (t\sigma)\gamma$ holds.
3. This follows from closure under contexts of $>$.
4. Any substitution γ that respects ψ also respects φ by Lemma 38, hence $s\gamma > t\gamma$ because of $s >_{[\varphi]} t$. ◀

Proof of Lemma 21. Suppose a step $C[s\tau] \leftrightarrow_{\mathsf{CP}(\mathcal{R} \cup \mathcal{E}^>)} C[t\tau]$ between ground terms $C[s\tau]$ and $C[t\tau]$ uses a critical pair $s \approx t \, [\chi]$ in $\mathsf{CP}(\mathcal{R} \cup \mathcal{E}^>)$ originating from an overlap $\langle \ell_1 \rightarrow r_1 \, [\varphi_1], p, \ell_2 \rightarrow r_2 \, [\varphi_2] \rangle$ with most general unifier σ. Hence we have $\ell_1 \rightarrow r_1 \, [\varphi_1]$, $\ell_2 \rightarrow r_2 \, [\varphi_2] \in \mathcal{R} \cup \mathcal{E}^> \cup \mathcal{R}_{\mathsf{calc}}$, $s = \ell_2\sigma[r_1\sigma]_p$, $t = r_2\sigma$, and $\chi = \varphi_1\sigma \wedge \varphi_2\sigma$. Then there are $u_1 \approx v_1 \, [\psi_1]$ and $u_2 \approx v_2 \, [\psi_2]$ in $\mathcal{R} \cup \mathcal{E} \cup \mathcal{R}_{\mathsf{calc}}$ and a substitution γ such that $\ell_1 \rightarrow r_1 \, [\varphi_1] = (u_1 \approx v_1 \, [\psi_1])\gamma$ and $\ell_2 \rightarrow r_2 \, [\varphi_2] = (u_2 \approx v_2 \, [\psi_2])\gamma$ (assuming that equations and rules in $\mathcal{R} \cup \mathcal{E} \cup \mathcal{R}_{\mathsf{calc}}$ are renamed apart).

We distinguish two cases. First, suppose $p \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(u_2)$. We have $u_2\gamma\sigma = u_2\gamma\sigma[u_1\gamma\sigma]$, so $u_2|_p$ and u_1 must be unifiable. Let ρ be their most general unifier, so there is some substitution δ such that $\gamma\sigma = \rho\delta$. Since $\chi\tau = (\varphi_1 \wedge \varphi_2)\sigma\tau = (\psi_1 \wedge \psi_2)\gamma\sigma\tau$ is valid, $(\psi_1 \wedge \psi_2)\gamma$ is satisfiable. So there is an overlap $\langle u_1 \approx v_1 \, [\psi_1], p, u_2 \approx v_2 \, [\psi_2] \rangle$. Moreover, since τ respects χ, the substitution $\sigma\tau$ respects $\varphi_1 = \psi_1\gamma$. If $u_1 \approx v_1 \, [\psi_1] \in \mathcal{R} \cup \mathcal{E}^>$ then we have $u_1\gamma >_{[\psi_1\gamma]} v_1\gamma$ and hence $u_1\gamma\sigma\tau > v_1\gamma\sigma\tau$; if $u_1 \approx v_1 \, [\psi_1] \in \mathcal{R}_{\mathsf{calc}}$ then $u_1\gamma\sigma\tau \geqslant v_1\gamma\sigma\tau$ by the properties of a reduction pair. In either case $v_1\gamma >_{[\psi_1\gamma]} u_1\gamma$ cannot hold because $>$ is well-founded. Similarly, $v_2\gamma >_{[\psi_2\gamma]} u_2\gamma$ cannot hold. So the overlap gives rise to a constrained extended critical pair $u_2\rho[v_1\rho]_p \approx v_2\rho \, [\psi_1\gamma \wedge \psi_2\gamma]$, and we have a step

$$C[s\tau] = C[u_2[v_1]_p\gamma\sigma\tau] = C[u_2\rho[v_1\rho]_p\delta\tau] \xleftrightarrow[\mathsf{CP}^>(\mathcal{R} \cup \mathcal{E})]{} C[v_2\rho\delta\tau] = C[v_2\gamma\sigma\tau] = C[t\tau]$$

Second, if $p \notin \mathcal{P}\mathsf{os}_{\mathcal{F}}(u_2)$ then the peak $u_2\gamma[v_1\gamma]_p \leftarrow u_2\gamma[u_1\gamma]_p = u_2\gamma \rightarrow v_2\gamma$ forms a variable overlap between $u_1 \approx v_1 \, [\psi_1]$ and $u_2 \approx v_2 \, [\psi_2]$. There are positions p' and q such that $u_2|_{p'}$ is some variable x and $\gamma(x)|_q = u_1\gamma$. Note that $u_2 \approx v_2 \, [\psi_2]$ cannot be a calculation rule since $\gamma(x) \notin \mathcal{V}\mathsf{al}$. Let γ' be the substitution defined by $\gamma'(x) = \gamma(x)[v_1\gamma]_q$ and $\gamma'(y) = \gamma(y)$

for all $y \neq x$. Then x cannot occur in ψ_2 since $\gamma(y) \in \mathcal{V}$al for all $y \in \mathcal{V}$ar(ψ_2), but left-hand sides are headed by non-theory symbols. Therefore γ' respects ψ_2 and thus we obtain

$$u_2\gamma[v_1\gamma]_p \xrightarrow[\ell_1 \to r_1 \ [\varphi_1]]{*} u_2\gamma' \xleftrightarrow[u_2 \approx v_2 \ [\psi_2]]{} v_2\gamma' \xleftarrow[\ell_1 \to r_1 \ [\varphi_1]]{*} v_2\gamma$$

Since $u_2\gamma$ and $v_2\gamma$ are ground, $u_2 \approx v_2 \ [\psi_2] \in \mathcal{R} \cup \mathcal{E}^\pm$ and $>$ is complete for $\mathcal{R} \cup \mathcal{E}$, either $u_2\gamma' \to_{\mathcal{R}\cup\mathcal{E}>} v_2\gamma'$ or $v_2\gamma' \to_{\mathcal{R}\cup\mathcal{E}>} u_2\gamma'$ must hold.   ◄

Proof of Lemma 26. We perform a case distinction on the applied inference rule.

- Suppose deduce was applied and there is a step $C[s\sigma] \leftrightarrow C[t\sigma]$ using $s \approx t \ [\varphi] \in \mathcal{E}'$. The substitution σ must respect φ. From $\langle u, u \rangle \ [\varphi] \to_{\mathcal{R}\cup\mathcal{E}^\pm} \langle s, u \rangle \ [\varphi]$ and Lemma 8(2) we obtain a substitution γ_1 such that $u\gamma_1 \to_{\mathcal{R}\cup\mathcal{E}^\pm} s\sigma$ and $u\gamma_1 = u\sigma$, and from $\langle u, u \rangle \ [\varphi] \to_{\mathcal{R}\cup\mathcal{E}^\pm} \langle u, t \rangle \ [\varphi]$ a substitution γ_2 such that $u\gamma_2 \to_{\mathcal{R}\cup\mathcal{E}^\pm} t\sigma$ and $u\gamma_2 = u\sigma$. We have

$$C[s\sigma] \ _{\mathcal{R}\cup\mathcal{E}^\pm}\!\!\leftarrow C[u\gamma_1] = C[u\sigma] = C[u\gamma_2] \to_{\mathcal{R}\cup\mathcal{E}^\pm} C[t\sigma]$$

  and thus $C[s\sigma] \leftrightarrow^*_{\mathcal{E}\cup\mathcal{R}} C[t\sigma]$.
- If compose was applied then $\langle s, t \rangle \ [\varphi] \to_{\mathcal{R}\cup\mathcal{E}>} \langle s, u \rangle \ [\psi]$ and hence $\langle t, s \rangle \ [\varphi] \to_{\mathcal{R}\cup\mathcal{E}>} \langle u, s \rangle \ [\psi]$. If there is a step $C[u\sigma] \leftarrow C[s\sigma]$ then $C[s\sigma] \leftrightarrow^2_{\mathcal{E}\cup\mathcal{R}} C[u\sigma]$ by Lemma 9(2).
- Next, suppose simplify was applied so we have $\langle s, t \rangle \ [\varphi] \to_{\mathcal{R}\cup\mathcal{E}>} \langle u, t \rangle \ [\psi]$. For a step $C[s\sigma] \leftrightarrow C[u\sigma]$ using $s \approx u \ [\psi]$ we thus have $C[s\sigma] \leftrightarrow^2_{\mathcal{E}\cup\mathcal{R}} C[u\sigma]$ by Lemma 9(2).
- If collapse was applied we have $\langle t, s \rangle \ [\varphi] \to_{\mathcal{R}\cup\mathcal{E}>} \langle u, s \rangle \ [\psi]$. If there is a step $C[u\sigma] \leftrightarrow C[s\sigma]$ using $u \approx s \ [\psi]$ then $C[s\sigma] \leftrightarrow^2_{\mathcal{E}\cup\mathcal{R}} C[u\sigma]$ follows again from Lemma 9(2).
- The case of orient is trivial, and in the case of delete there is nothing to show. Also the split cases are easy since substitutions respecting $\varphi \wedge \psi$ (or $\varphi \wedge \neg\psi$) also respect φ.   ◄

ProTeM: A Proof Term Manipulator

Christina Kohl

Department of Computer Science, University of Innsbruck, Austria
christina.kohl@uibk.ac.at
🆔 https://orcid.org/0000-0002-8470-2485

Aart Middeldorp

Department of Computer Science, University of Innsbruck, Austria
aart.middeldorp@uibk.ac.at
🆔 https://orcid.org/0000-0001-7366-8464

―――― **Abstract** ――――――――――――――――――――――――――――――――――――――

Proof terms are a useful concept for reasoning about computations in term rewriting. Human calculation with proof terms is tedious and error-prone. We present ProTeM, a new tool that offers support for manipulating proof terms that represent multisteps in left-linear rewrite systems.

1 Introduction

Proof terms represent computations in term rewriting. They were introduced by van Oostrom and de Vrijer for first-order left-linear rewrite systems to study equivalence of reductions in [12] and [9, Chapter 8]. Extensions to higher-order rewriting and infinitary rewriting are reported in [1] and [5], respectively. Hirokawa and the second author used proof terms for confluence analysis of left-linear rewrite systems [2, 3].

Our motivation for studying proof terms is to close an important gap between proofs produced by automatic confluence checkers and *certified* proofs. Numerous confluence criteria described in the literature have been formalized in IsaFoR, a large Isabelle/HOL library for term rewriting, see [7] for a recent overview. This includes the well-known result of Huet [4] stating that a left-linear rewrite system is confluent if its critical pairs are closed by a parallel step [8]. Its extension to multisteps (also called development steps) by van Oostrom [11] thus far escaped all attempts to obtain a formalized proof. The picture proof in [11] conveys the intuition but is very hard to formalize in a modern proof assistant. We believe that proof terms together with residual theory [9, Section 8.7] will help to close the gap.

Calculations with proof terms are tedious and error-prone to do by hand, which is why we developed ProTeM. Besides providing basic operations for manipulating proof terms that represent multisteps in left-linear rewrite systems, like join and residual, ProTeM supports new operations on proof terms that are required for a formalized proof of the main result of

[11]. The latter include an inductive definition for computing the amount of overlap and a function that returns the critical overlaps between co-initial proof terms.

In the next section we recall proof terms and introduce new operations for measuring overlap between two proof terms. The web interface of ProTeM is described in Section 3 and in Section 4 we present some implementation details. We conclude in Section 5 with ideas for future extensions of ProTeM.
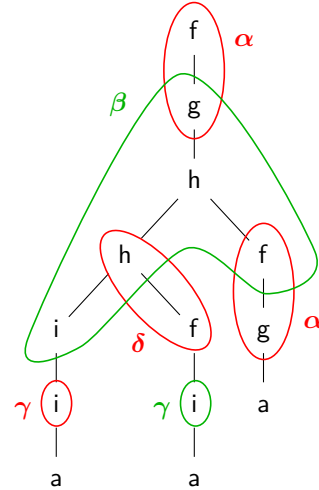
2 Proof Terms

Proof terms are built from function symbols, variables, and rule symbols. The latter represent rewrite rules and have a fixed arity which is the number of different variables in the represented rule. We use Greek letters as rule symbols. In this section we present the operations on proof terms that are implemented in ProTeM. The following example will be used to illustrate various definitions.

▶ **Example 1.** Consider the rewrite rules

$$\alpha: \qquad\qquad \mathsf{f}(\mathsf{g}(x)) \to \mathsf{g}(\mathsf{h}(x,\mathsf{i}(\mathsf{a})))$$
$$\beta: \quad \mathsf{g}(\mathsf{h}(\mathsf{h}(\mathsf{i}(x),y),\mathsf{f}(z))) \to \mathsf{h}(\mathsf{h}(y,y),\mathsf{f}(z))$$
$$\gamma: \qquad\qquad\qquad \mathsf{i}(x) \to x$$
$$\delta: \qquad\qquad \mathsf{h}(x,\mathsf{f}(y)) \to \mathsf{h}(\mathsf{i}(\mathsf{f}(y)),\mathsf{f}(y))$$
$$\varepsilon: \qquad\qquad \mathsf{g}(\mathsf{h}(x,y)) \to \mathsf{h}(x,y)$$

and the term $s = \mathsf{f}(\mathsf{g}(\mathsf{h}(\mathsf{h}(\mathsf{h}(\mathsf{i}(\mathsf{i}(\mathsf{a})),\mathsf{f}(\mathsf{i}(\mathsf{a}))),\mathsf{f}(\mathsf{g}(\mathsf{a}))))))$. By marking certain redexes in s we obtain the two proof terms

$$A = \alpha(\mathsf{h}(\delta(\mathsf{i}(\gamma(\mathsf{a})),\mathsf{i}(\mathsf{a})),\alpha(\mathsf{a})))$$
$$B = \mathsf{f}(\beta(\mathsf{i}(\mathsf{a}),\mathsf{f}(\gamma(\mathsf{a})),\mathsf{g}(\mathsf{a})))$$

This situation is illustrated on the right, where the redexes in A are indicated in red and those in B in green.

If α is a rule symbol then $\mathsf{lhs}(\alpha)$ ($\mathsf{rhs}(\alpha)$) denotes the left-hand (right-hand) side of the rewrite rule represented by α. Furthermore $\mathsf{var}(\alpha)$ denotes the list $(x_1,\dots,x_n)$ of variables appearing in α in some fixed order. The length of this list is the arity of α. Given a rule symbol α with $\mathsf{var}(\alpha) = (x_1,\dots,x_n)$ and terms $t_1,\dots,t_n$, we write $\langle t_1,\dots,t_n\rangle_\alpha$ for the substitution $\{x_i \mapsto t_i \mid 1 \leqslant i \leqslant n\}$. A proof term A witnesses a multistep from its source $\mathsf{src}(A)$ to its target $\mathsf{tgt}(A)$, which are computed as follows:

$$\mathsf{src}(x) = \mathsf{tgt}(x) = x$$
$$\mathsf{src}(f(A_1,\dots,A_n)) = f(\mathsf{src}(A_1),\dots,\mathsf{src}(A_n))$$
$$\mathsf{src}(\alpha(A_1,\dots,A_n)) = \mathsf{lhs}(\alpha)\langle\mathsf{src}(A_1),\dots,\mathsf{src}(A_n)\rangle_\alpha$$
$$\mathsf{tgt}(f(A_1,\dots,A_n)) = f(\mathsf{tgt}(A_1),\dots,\mathsf{tgt}(A_n))$$
$$\mathsf{tgt}(\alpha(A_1,\dots,A_n)) = \mathsf{rhs}(\alpha)\langle\mathsf{tgt}(A_1),\dots,\mathsf{tgt}(A_n)\rangle_\alpha$$

Here f is an n-ary function symbol. Proof terms A and B are *co-initial* if they have the same source. We define the *orthogonality* predicate $A \perp B$ by the following clauses:

$$x \perp x$$

$$f(A_1, \ldots, A_n) \perp f(B_1, \ldots, B_n) \iff A_i \perp B_i \text{ for all } 1 \leqslant i \leqslant n$$

$$\alpha(A_1, \ldots, A_n) \perp \mathsf{lhs}(\alpha)\langle B_1, \ldots, B_n\rangle_\alpha \iff A_i \perp B_i \text{ for all } 1 \leqslant i \leqslant n$$

$$\mathsf{lhs}(\alpha)\langle A_1, \ldots, A_n\rangle_\alpha \perp \alpha(B_1, \ldots, B_n) \iff A_i \perp B_i \text{ for all } 1 \leqslant i \leqslant n$$

In all other cases $A \perp B$ is false. Next we recall the join $(A \sqcup B)$ and residual (A / B) operations on co-initial proof terms:

$$x \sqcup x = x / x = x$$

$$f(A_1, \ldots, A_n) \sqcup f(B_1, \ldots, B_n) = f(A_1 \sqcup B_1, \ldots, A_n \sqcup B_n)$$

$$\alpha(A_1, \ldots, A_n) \sqcup \alpha(B_1, \ldots, B_n) = \alpha(A_1 \sqcup B_1, \ldots, A_n \sqcup B_n)$$

$$\alpha(A_1, \ldots, A_n) \sqcup \mathsf{lhs}(\alpha)\langle B_1, \ldots, B_n\rangle_\alpha = \alpha(A_1 \sqcup B_1, \ldots, A_n \sqcup B_n)$$

$$\mathsf{lhs}(\alpha)\langle A_1, \ldots, A_n\rangle_\alpha \sqcup \alpha(B_1, \ldots, B_n) = \alpha(A_1 \sqcup B_1, \ldots, A_n \sqcup B_n)$$

$$f(A_1, \ldots, A_n) / f(B_1, \ldots, B_n) = f(A_1 / B_1, \ldots, A_n / B_n)$$

$$\alpha(A_1, \ldots, A_n) / \alpha(B_1, \ldots, B_n) = \mathsf{rhs}(\alpha)\langle A_1 / B_1, \ldots, A_n / B_n\rangle_\alpha$$

$$\alpha(A_1, \ldots, A_n) / \mathsf{lhs}(\alpha)\langle B_1, \ldots, B_n\rangle_\alpha = \alpha(A_1 / B_1, \ldots, A_n / B_n)$$

$$\mathsf{lhs}(\alpha)\langle A_1, \ldots, A_n\rangle_\alpha / \alpha(B_1, \ldots, B_n) = \mathsf{rhs}(\alpha)\langle A_1 / B_1, \ldots, A_n / B_n\rangle_\alpha$$

These are partial operations. The next operation that we define on proof terms is *deletion* $A - B$, which is used to remove steps from a multistep:

$$x - x = x$$

$$f(A_1, \ldots, A_n) - f(B_1, \ldots, B_n) = f(A_1 - B_1, \ldots, A_n - B_n)$$

$$\alpha(A_1, \ldots, A_n) - \alpha(B_1, \ldots, B_n) = \mathsf{lhs}(\alpha)\langle A_1 - B_1, \ldots, A_n - B_n\rangle_\alpha$$

$$\alpha(A_1, \ldots, A_n) - \mathsf{lhs}(\alpha)\langle B_1, \ldots, B_n\rangle_\alpha = \alpha(A_1 - B_1, \ldots, A_n - B_n)$$

Like join and residual, deletion is a partial operation.

▶ **Example 2.** The proof terms A and B in Example 1 are not orthogonal. Let $C = B - \mathsf{f}(\beta(\mathsf{i}(\mathsf{a}), \mathsf{f}(\mathsf{i}(\mathsf{a})), \mathsf{g}(\mathsf{a}))) = \mathsf{f}(\mathsf{g}(\mathsf{h}(\mathsf{h}(\mathsf{i}(\mathsf{i}(\mathsf{a})), \mathsf{f}(\gamma(\mathsf{a})), \mathsf{f}(\mathsf{g}(\mathsf{a}))))))$. We have $A \perp C$. Moreover,

$$A / C = \alpha(\mathsf{h}(\delta(\mathsf{i}(\gamma(\mathsf{a})), \mathsf{a}), \alpha(\mathsf{a})))$$

$$C / A = \mathsf{g}(\mathsf{h}(\mathsf{h}(\mathsf{h}(\mathsf{i}(\mathsf{f}(\mathsf{f}(\gamma(\mathsf{a}))), \mathsf{f}(\gamma(\mathsf{a}))), \mathsf{g}(\mathsf{h}(\mathsf{a}, \mathsf{i}(\mathsf{a})))), \mathsf{i}(\mathsf{a})))$$

An important concept in the correctness proof of the confluence theorem in [11] is the amount of overlap between two multisteps. Below we present an inductive definition for measuring the overlap between co-initial proof terms. It is based on a special labeling of the source of a proof term. We write $\mathsf{lhs}^\sharp(\alpha)$ for the result of labeling every function symbol in $\mathsf{lhs}(\alpha)$ with α as well as the distance to the root of α: $\mathsf{lhs}^\sharp(\alpha) = \varphi(\mathsf{lhs}(\alpha), \alpha, 0)$ with

$$\varphi(t, \alpha, i) = \begin{cases} t & \text{if } t \in \mathcal{V} \\ f_{\alpha^i}(\varphi(t_1, \alpha, i+1), \ldots, \varphi(t_n, \alpha, i+1)) & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

The mapping $\mathsf{src}^\sharp$ computes the labeled source of a proof term:

$$\mathsf{src}^\sharp(x) = x$$

$$\mathsf{src}^\sharp(f(A_1, \ldots, A_n)) = f(\mathsf{src}^\sharp(A_1), \ldots, \mathsf{src}^\sharp(A_n))$$

$$\mathsf{src}^\sharp(\alpha(A_1, \ldots, A_n)) = \mathsf{lhs}^\sharp(\alpha)\langle \mathsf{src}^\sharp(A_1), \ldots, \mathsf{src}^\sharp(A_n)\rangle_\alpha$$

Given two co-initial proof terms A and B, the following function computes a single labeled term in which all function symbols corresponding to redex patterns in A and B are marked:

$$\mathsf{merge}(A, B) = \mathsf{merge}'(\mathsf{src}^\sharp(A), \mathsf{src}^\sharp(B))$$

with $\mathsf{merge}'(s, t) = s$ for $s, t \in \mathcal{V}$ and $\mathsf{merge}'(s, t) = f_{ab}(\mathsf{merge}'(s_1, t_1), \ldots, \mathsf{merge}'(s_n, t_n))$ if $s = f_a(s_1, \ldots, s_n)$ and $t = f_b(t_1, \ldots, t_n)$. Here we identify an unlabeled function symbol f with f_-. The merge function is used to measure the amount of overlap between co-initial proof terms: $\blacktriangle(A, B) = \mathsf{measure}(\mathsf{merge}(A, B))$ with $\mathsf{measure}(u) = 0$ if $u \in \mathcal{V}$ and

$$\mathsf{measure}(f_{a^k b^l}(u_1, \ldots, u_n)) = \begin{cases} 1 + \sum_{i=1}^n \mathsf{measure}(u_i) & \text{if } a^k \neq - \text{ and } b^l \neq - \\ \sum_{i=1}^n \mathsf{measure}(u_i) & \text{otherwise} \end{cases}$$

Finally the $\mathsf{overlaps}$ function collects all pairs of overlapping redexes in co-initial proof terms:

$$\mathsf{overlaps}(A, B) = \left\{ (p, \alpha, q, \beta) \ \middle| \ \begin{array}{l} p, q \in \mathcal{P}\mathsf{os}_\mathcal{F}(u), \ \ell_1(u(p)) = \alpha^0, \ \ell_2(u(q)) = \beta^0, \text{ and either} \\ p \leqslant q \text{ and } \ell_1(u(q)) = \alpha^{|q \backslash p|} \text{ or } q < p \text{ and } \ell_2(u(p)) = \beta^{|p \backslash q|} \end{array} \right\}$$

Here $u = \mathsf{merge}(A, B)$ and $\mathcal{P}\mathsf{os}_\mathcal{F}(u)$ is the set of function positions in u. The functions ℓ_1 and ℓ_2 extract the first and second label of a labeled function symbol: $\ell_1(f_{ab}) = a$ and $\ell_2(f_{ab}) = b$. The condition $\ell_1(u(q)) = \alpha^{|q \backslash p|}$ in the first case of the definition of $\mathsf{overlaps}(A, B)$ ensures that $q \backslash p$ is a position in $\mathsf{lhs}(\alpha)$.

▶ **Example 3.** For the proof terms in Example 1 we have

$$\mathsf{merge}(A, B) = \mathsf{f}_{\boldsymbol{\alpha}^0 -}(\mathsf{g}_{\boldsymbol{\alpha}^1 \boldsymbol{\beta}^0}(\mathsf{h}_{-\boldsymbol{\beta}^1}(\mathsf{h}_{\boldsymbol{\delta}^0 \boldsymbol{\beta}^2}(\mathsf{i}_{-\boldsymbol{\beta}^3}(\mathsf{i}_{\boldsymbol{\gamma}^0 -}(\mathsf{a})), \mathsf{f}_{\boldsymbol{\delta}^1 -}(\mathsf{i}_{-\boldsymbol{\gamma}^0}(\mathsf{a}))), \mathsf{f}_{\boldsymbol{\alpha}^0 \boldsymbol{\beta}^2}(\mathsf{g}_{\boldsymbol{\alpha}^1 -}(\mathsf{a})))))$$

and $\blacktriangle(A, B) = 3$. Here a abbreviates a_{--}. Furthermore, $\mathsf{overlaps}(A, B)$ consists of the tuples $(\epsilon, \boldsymbol{\alpha}, 1, \boldsymbol{\beta})$, $(111, \boldsymbol{\delta}, 1, \boldsymbol{\beta})$, and $(112, \boldsymbol{\alpha}, 1, \boldsymbol{\beta})$.

3   Web Interface

In this section we will first give a brief overview of the main parts of ProTeM's user interface, subsequently we will describe all features in more detail. The web interface of ProTeM can be accessed at

<div align="center">

`http://informatik-protem.uibk.ac.at/`

</div>

The layout of our application is displayed in Figure 1. At the center of the screen we have a large area for displaying the history of commands a user has entered (on the left), together with result output corresponding to these commands (on the right). We also offer the possibility to export the commands and results of the current session as a simple text file via the "Session Log" button underneath the output area. Below that there is a smaller panel where all rules of the currently loaded term rewrite system are displayed. At the bottom of the screen we have a command line with several buttons above it, that help users enter unusual symbols such as Greek letters for rule symbols or the ⊥ symbol for the orthogonality predicate on proof terms. To the left of the screen we have a sidebar that gives an overview of the syntax that is used for commands. In the navigation bar at the top right corner of the screen we have a link leading to a help page with details about every component and feature of ProTeM.
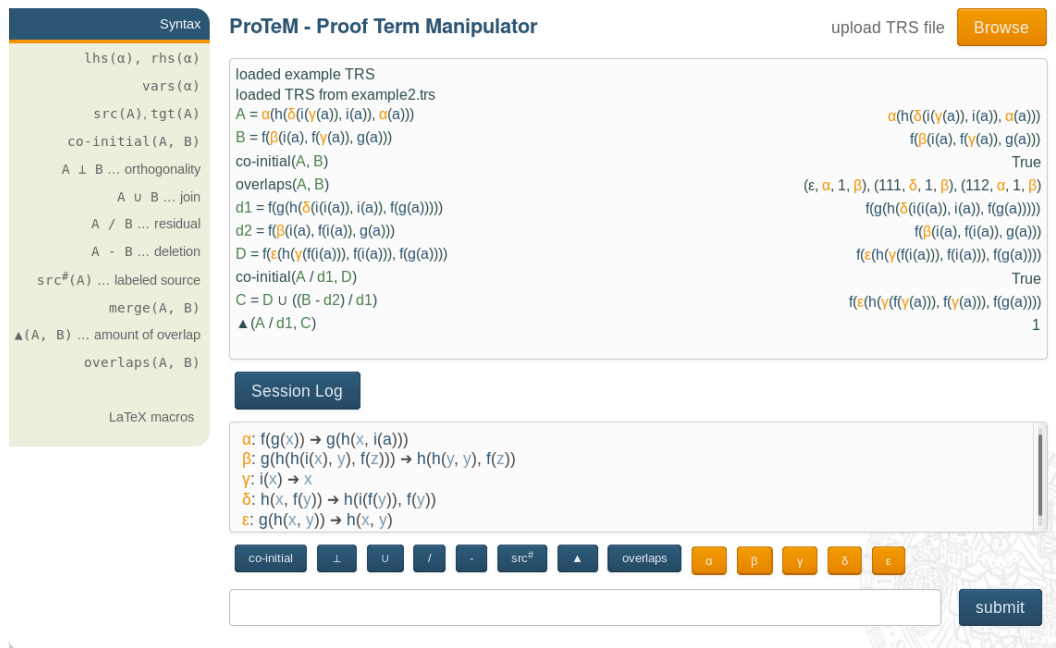
Figure 1 Screenshot of a ProTeM session.

3.1 Uploading a Term Rewrite System

When first opening the website, a simple example rewrite system is loaded per default. Users can upload their own rewrite systems from `.trs` files. The files need to correspond to a simplified form of the standard TRS-format as described in [6], where only the `VAR` and `RULES` sections are taken into account. Additionally the rule symbols ProTeM should use can be specified in the file by prepending each rule with its corresponding symbol followed by a colon. For reference, an example `.trs` file is available in the help section of the tool. If one or more rules have no specified rule symbols, ProTeM chooses a new Greek letter for each rule, starting from α. In cases where there are more than 24 rules, ProTeM begins to append digits to each Greek letter, e.g. $\alpha 1$, $\beta 1$, $\gamma 1$, ... When uploading a new rewrite system, the buttons above the command line will automatically change according to the new rule symbols.

3.2 Commands

There are two types of commands available, one are assignments, the other computations on proof terms. Assignments have syntax `id = proofterm` where `id` can be any string and `proofterm` any valid proof term. Notably it is possible to use nested expressions in assignments (e.g. $C = D \sqcup ((B - \mathsf{d2})/\mathsf{d1})$, see also Figure 1). Commands for rule symbols are $\mathtt{lhs}(\alpha)$, $\mathtt{rhs}(\alpha)$ and $\mathtt{vars}(\alpha)$ where α can be any rule symbol used in the current TRS. Commands for proof terms include all operations described in Section 2 and their nested applications. The syntax of these operations is listed in the sidebar of our application.

Commands have to be entered into the text field at the bottom of the screen. The blue buttons above it can be used to enter special symbols that are used for some of the commands (like $\perp$ for orthogonality, or ▲ for measuring the amount of overlap between two proof terms). In addition there is one orange button for each currently used rule symbol.

When pressing one of the buttons, the corresponding symbol appears in the command line, with the focus returning immediately to the text field itself so that the user can carry on typing. A command can be submitted either by pressing enter or by using the "Submit" button. If the command line contains a valid command, it will be sent to the server and executed. The result will then be displayed in the output area above. If the command was not valid (e.g. trying to assign the result of an undefined operation), an error message will be displayed (see Figure 2 in Appendix A).

3.3   Export to LaTeX

A proof term or labeled proof term can be exported as a LaTeX string. To correctly insert proof terms from ProTeM into a LaTeX document it is first necessary to add the required macros. These define colors and provide support for UTF8 encoding of Greek letters. In particular we define three new commands `\pfun`, `\pvar`, `\prule` which define the representations of function symbols, variables and rule symbols respectively. The macros can be downloaded by clicking on the "LaTeX macros" entry in the sidebar. Clicking on any proof term in the output area will open a popup view, which contains a text field with the LaTeX representation of that proof term (see Figure 3 in Appendix A). It can then be copy-and-pasted into any document.

4   Implementation Details

The core functionality of ProTeM is written in Scala. For the web component we used the Vaadin framework [10]. Vaadin is a Java web application framework that makes it easier for developers who don't have much experience with web technologies, such as JavaScript, HTML and HTTP requests, to design responsive and interactive web applications. Vaadin allows developers to write all required code in pure Java (or any other language that runs on the JVM). Applications can also be extended with custom HTML or JavaScript and themed with CSS. From a technological point of view the UI logic of a Vaadin application runs as a Java Servlet in a Java application server. On the client side Vaadin uses JavaScript to render the user interface in the browser and communicate user events to the server. All communication is automated and makes heavy use of AJAX (Asynchronous JavaScript and XML) to make applications as responsive as possible. An additional benefit for our particular application was that Vaadin automatically stores the state of each user session (as long as the browser window is open), so that we can provide users with an interactive interface and still call our Scala functions on the server for all computations on proof terms.

5   Conclusion

In this paper we presented ProTeM, a tool that supports operations on proof terms that represent multisteps in first-order left-linear rewrite systems. There are several possibilities to extend the functionality of ProTeM. First of all, adding a composition operation to the language of proof terms allows to represent rewrite sequences that are not single multisteps. Equivalence testing and normalization become then interesting questions. Also one could ask the tool to compute proof terms that represent a given rewrite sequence. Another useful extension will be automatic support for visualizing co-initial proof terms, like the figure in Example 1. Dropping the left-linearity requirement will be a challenging task, which requires the development of new theory.

───── **References** ─────

1 H.J. Sander Bruggink. *Equivalence of Reductions in Higher-Order Rewriting*. PhD thesis, Utrecht University, 2008.

2 Nao Hirokawa and Aart Middeldorp. Decreasing diagrams and relative termination. *Journal of Automated Reasoning*, 47(4):481–501, 2011. `doi:10.1007/s10817-011-9238-x`.

3 Nao Hirokawa and Aart Middeldorp. Commutation via relative termination. In *Proc. 2nd International Workshop on Confluence*, pages 29–33, 2013.

4 Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980. `doi:10.1145/322217.322230`.

5 Carlos Lombardi, Alejandro Ríos, and Roel de Vrijer. Proof terms for infinitary rewriting. In *Proceedings of the Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications*, volume 8560 of *Lecture Notes in Computer Science (Advanced Research in Computing and Software Science)*, pages 303–318, 2014. `doi:10.1007/978-3-319-08918-8_21`.

6 Claude Marché, Albert Rubio, and Hans Zantema. Termination problem data base: Format of input files. `https://www.lri.fr/~marche/tpdb/format.html`. Accessed: 2018-17-01.

7 Julian Nagele. *Mechanizing Confluence*. PhD thesis, University of Innsbruck, 2017.

8 Julian Nagele and Aart Middeldorp. Certification of classical confluence results for left-linear term rewrite systems. In *Proc. 7th International Conference on Interactive Theorem Proving*, volume 9807 of *Lecture Notes in Computer Science*, pages 290–306, 2016. `doi:10.1007/978-3-319-43144-4_18`.

9 Terese, editor. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

10 Vaadin framework 8. `https://vaadin.com/docs/v8/framework/introduction/intro-overview.html`. Accessed: 2018-17-01.

11 Vincent van Oostrom. Developing developments. *Theoretical Computer Science*, 175(1):159–181, 1997. `doi:10.1016/S0304-3975(96)00173-9`.

12 Vincent van Oostrom and Roel de Vrijer. Four equivalent equivalences of reductions. In *Proc. 2nd International Workshop on Reduction Strategies in Rewriting and Programming*, volume 70(6) of *Electronic Notes in Theoretical Computer Science*, pages 21–61, 2002. `doi:10.1016/S1571-0661(04)80599-1`.

A   Additional Screenshots

This appendix contains the screenshots referred to in Sections 3.2 and 3.3.

Figure 2 An invalid assignment; the join operation of these two proof terms is not defined.

Figure 3 Popup view containing the LaTeX representation of a labeled proof term.

Confluence Competition 2018

Takahito Aoto[1]
Faculty of Engineering, Niigata University, Japan
aoto@ie.niigata-u.ac.jp
 https://orcid.org/0000-0003-0027-0759

Makoto Hamana[2]
Department of Computer Science, Gunma University, Japan
hamana@cs.gunma-u.ac.jp
 https://orcid.org/0000-0002-3064-8225

Nao Hirokawa[3]
School of Information Science, JAIST, Japan
hirokawa@jaist.ac.jp
 https://orcid.org/0000-0002-8499-0501

Aart Middeldorp[4]
Department of Computer Science, University of Innsbruck, Austria
aart.middeldorp@uibk.ac.at
 https://orcid.org/0000-0001-7366-8464

Julian Nagele
School of Electronic Engineering and Computer Science, Queen Mary University of London, UK
j.nagele@qmul.ac.uk
 https://orcid.org/0000-0002-4727-4637

Naoki Nishida
Graduate School of Informatics, Nagoya University, Japan
nishida@i.nagoya-u.ac.jp
 https://orcid.org/0000-0001-8697-4970

Kiraku Shintani
School of Information Science, JAIST, Japan
s1820017@jaist.ac.jp
 https://orcid.org/0000-0002-2986-4326

Harald Zankl
Innsbruck, Austria
hzankl@gmail.com
 https://orcid.org/0000-0003-2516-4223

── **Abstract** ──────────────────────────────

We report on the 2018 edition of the Confluence Competition, a competition of software tools that aim to (dis)prove confluence and related properties of rewrite systems automatically.

1   Confluence Competition

The annual Confluence Competition (CoCo)[5] has driven the development of techniques for (dis)proving confluence and related properties of a variety of rewrite formalisms automatically. Starting in 2012 with 4 tools competing in 2 categories, CoCo has grown steadily to 11 categories with 11 tools in 2017, and several tools ran in multiple categories.

CoCo is executed on the dedicated high-end cross-community competition platform *StarExec* [3]. A speciality of CoCo is that the whole competition is conducted within one slot at a conference or workshop (IWC in 2012–2017 and FSCD in 2018). The progress of the live competition is shared with the audience visually through the *LiveView* tool which interacts with StarExec. A screenshot of the LiveView of CoCo 2017 is shown in Figure 1.

2   Categories

CoCo supports two kinds of categories, *competition* and *demonstration* categories. The latter are one-time events for demonstrating new rewrite formats or properties. These can be requested until 2 months before a competition. *Competition* categories run also in future editions of CoCo. These can be requested until 6 months prior to the competition, in order to allow the CoCo steering committee to make a well-informed decision on the format (precise syntax as well as semantics) of the new categories and to extend the Confluence Problems database (Cops) accordingly. In CoCo 2018, we have the following 11 competition categories:

TRS/CTRS/HRS These three categories are about confluence of three important formalisms of rewriting, namely, *first-order term rewriting* (TRS), *conditional term rewriting* (CTRS), and *higher-order rewriting* (HRS).

CPF-TRS/CPF-CTRS These two categories are for *certified* confluence proofs. Participating tools must generate certificates that are checked by an independent certifier.

GCR This category is about *ground* confluence of many-sorted term rewrite systems.

NFP/UNC/UNR These categories are about confluence-related properties of first-order term rewrite systems, namely, *the normal form property* (NFP), *unique normal forms with respect to conversion* (UNC), and *unique normal forms with respect to reduction* (UNR).

CPF This category is the combination of the CPF-TRS and CPF-CTRS categories, evaluating the overall power of tools that generate certified confluence (dis)proofs.

UN This category is the combination of the NFP, UNC, and UNR categories. Tools compete to prove the strongest property among these three.

As GCR and NFP/UNC/UNR are new categories introduced in the last competition, we provide some more details. The other categories are described in the CoCo 2015 report [1]. Applications based on initial algebra semantics often rely on confluence of well-sorted ground

---

[5] http://coco.nue.ie.niigata-u.ac.jp/

CoCo 2017

TRS

problems: 100
solvers: **ACP ver.0.51 2017 CoLL-Saigawa CSI**

ACP ver.0.51 2017 (100 of 100):   score: **75.00%**
YES:50 NO:25 MAYBE:25

CoLL-Saigawa (100 of 100):   score: **50.00%**
YES:35 NO:15 MAYBE:50

CSI (100 of 100):   score: **84.00%**
YES:57 NO:27 MAYBE:16

CTRS

problems: 100
solvers: **CO3 ConCon 1.5**

CO3 (100 of 100):   score: **55.00%**
YES:38 NO:17 MAYBE:45

ConCon 1.5 (100 of 100):   score: **74.00%**
YES:51 NO:23 MAYBE:26

HRS

problems: 77
solvers: **ACPH ver.0.02 2017 CSI^ho SOL-Ver1c_2017**

ACPH ver.0.02 2017 (77 of 77):   score: **79.22%**
YES:48 NO:13 MAYBE:16

CSI^ho (77 of 77):   score: **87.01%**
YES:55 NO:12 MAYBE:10

SOL-Ver1c_2017 (77 of 77):   score: **84.42%**
YES:54 NO:11 MAYBE:12

CPF-TRS

problems: 100
solvers: **ACP ver.0.51 and CeTA 2017 CSI+CeTA**

ACP ver.0.51 and CeTA 2017 (100 of 100): score: **33.00%**
YES:11 NO:22 MAYBE:67

CSI+CeTA (100 of 100):   score: **63.00%**
YES:36 NO:27 MAYBE:37

CPF-CTRS

problems: 100
solvers: **ConCon 1.5 + CeTA**

ConCon 1.5 + CeTA (100 of 100):   score: **68.00%**
YES:45 NO:23 MAYBE:32

GCR

problems: 64
solvers: **AGCP ver.0.03 FORT 1.0**

AGCP ver.0.03 (64 of 64):   score: **78.13%**
YES:15 NO:35 MAYBE:14

FORT 1.0 (64 of 64):   score: **25.00%**
YES:3 NO:13 MAYBE:48

NFP

problems: 60
solvers: **CSI FORT 1.0**

CSI (60 of 60):   score: **55.00%**
YES:4 NO:29 MAYBE:27

FORT 1.0 (60 of 60):   score: **25.00%**
YES:5 NO:10 MAYBE:45

UNC

problems: 60
solvers: **CSI FORT 1.0**

CSI (60 of 60):   score: **65.00%**
YES:10 NO:29 MAYBE:21

FORT 1.0 (60 of 60):   score: **25.00%**
YES:5 NO:10 MAYBE:45

UNR

problems: 60
solvers: **CSI FORT 1.0**

CSI (60 of 60):   score: **65.00%**
YES:10 NO:29 MAYBE:21

FORT 1.0 (60 of 60):   score: **25.00%**
YES:8 NO:7 MAYBE:45

This competition is finished

powered by **StarExec**

Figure 1 LiveView of CoCo 2017 upon completion.

terms, which is the reason why the GCR category deals with confluence of all well-sorted ground terms in a many-sorted term rewrite system. Uniqueness of normal forms also plays an important role in applications of rewriting. The notion is formalized in three different ways: A TRS satisfies NFP if $s \to^* t$ whenever $s \leftrightarrow^* t$ with t a normal form. A TRS satisfies UNC if $s = t$ whenever $s \leftrightarrow^* t$ with s and t normal forms. Finally, a TRS satisfies UNR if $s = t$ whenever $s \; ^*\!\leftarrow \cdot \to^* t$ with s and t normal forms. The properties GCR, NFP, UNC, and UNR are all weaker than confluence and the implications "NFP $\Rightarrow$ UNC $\Rightarrow$ UNR" hold.

3   Problems

Problems selected for CoCo originate from Cops, an online database of confluence problems.[6] Via its web interface, everyone can retrieve and download confluence problems, and also submit new problems. Figure 2 shows the submission interface of Cops. Submitted problems

---

[6] `http://cops.uibk.ac.at/`

◾ **Figure 2** The submission interface of Cops.

are reviewed by the CoCo steering committee and then integrated into Cops. We refer to the website and [2] for detailed information about Cops. Problem selection for CoCo is subject to the following constraints:

- Only problems stemming from the literature are considered. This includes papers presented at informal workshops like the International Workshop on Confluence (IWC) and PhD theses. The reason for this restriction is to avoid bias towards one particular tool or technique.
- For the GCR, NFP, UNC and UNR categories, only non-confluent problems are considered.
- For the CTRS and CPF-CTRS categories, only *oriented* conditional term rewrite systems *of type 3* are considered.
- The restriction to *pattern* rewrite systems in the HRS category for CoCo 2015–2017 has been removed in CoCo 2018.

Further selection details are available from the CoCo website.

For the live competition, 100 suitable problems are randomly selected for each category. In the demonstration categories, participating tool authors are requested to provide the problems for the competition.

4 Evaluation

Given a problem, participating tools must—in the first line of their output—answer `YES` or `NO` within 60 seconds; any other answer indicates that the tool could not determine the status of the problem. The winner of each category is determined by the total number of `YES`/`NO` answers. The combined UN category is an exception to this rule. The winner in that category is determined by summing up the points earned according to Table 1. Here, the

▪ **Table 1** Scoring in the UN category.

	none	UNR	UNC	NFP
none	0	3	4	5
¬NFP	3	4	5	
¬UNC	4	5		
¬UNR	5			

column corresponds to the strongest property proved by the tool, and the row corresponds to the weakest property refuted by the tool. For example, a tool that proves UNR, disproves NFP, but does not decide UNC, would score 4 points (row: ¬NFP, column: UNR).

Shortly after each competition, detailed competition results are made available on the CoCo website and integrated into Cops as metadata to indicate the statuses of problems in past competitions. If a tool has given a non-plausible answer, it is disqualified as a winner of the categories in which those answers are involved. Moreover, the records are corrected if such an erroneous answer is spotted after the live competition.

Most participating tools are available at the CoCo website. Moreover, CoCoWeb[7] [2] provides a convenient web interface to execute tools without local installation.

5 Outlook

We expect CoCo to grow with new categories and tools in the years ahead. Natural candidates are commutation, rewriting modulo AC, and nominal rewriting. We are planning to enhance the functionality of the LiveView tool. In particular, we want to have it recognize YES/NO conflicts among participating tools in real time. It would also be nice if the scores of the UN and CPF categories are computed and viewed in real time. Based on the metadata of (completed) competitions imported into Cops, a website[8] admits to view details of past competitions, including the output of tools. Importing the metadata in real time would make all details of the live competition immediately accessible.

References

1. T. Aoto, N. Hirokawa, J. Nagele, N. Nishida, and H. Zankl. Confluence Competition 2015. In *Proc. 25th CADE*, volume 9195 of *LNAI*, pages 101–104, 2015. `doi:10.1007/978-3-319-21401-6_5`.
2. N. Hirokawa, J. Nagele, and A. Middeldorp. Cops and CoCoWeb – Infrastructure for confluence tools. In *Proc. 9th IJCAR*, LNAI, 2018. To appear.
3. A. Stump, G. Sutcliffe, and C. Tinelli. StarExec: A cross-community infrastructure for logic solving. In *Proc. 7th IJCAR*, volume 8562 of *LNAI*, pages 367–373, 2014. `doi:10.1007/978-3-319-08587-6_28`.

---

[7] `http://cocoweb.uibk.ac.at/`
[8] `http://cops.uibk.ac.at/results/`