PETER THIEMANN

LaToKi: A Language Toolkit for Bottom-Up Evaluation of Functional Programms

Originalbeitrag erschienen in:
Maurice Bruynooghe (Hrsg.): Programming language implementation and logic programming : 4th international symposium ; proceedings.
Berlin [u.a.]: Springer, 1992, S. 481 - 482

LaToKi: A Language Toolkit for Bottom-Up Evaluation of Functional Programs

Peter Thiemann

Wilhelm Schickard Institut, Universität Tübingen, Sand 13, D-W7400 Tübingen, Germany

Abstract. LaToKi is a toolkit for experimentation with different implementations of recursion in strict functional programs. Its main emphasis is on the bottom-up evaluation of structural recursive defined programs. We have developed a technique that allows the evaluation of a wide subclass of structural recursive functions using a constant amount of control memory.

1 Introduction

The purpose of LaToKi is to provide an environment to compare different evaluation strategies for strict functional programs. We felt that such a toolkit was needed when we tried to compare the usual runtime stack implementation and its improvements (elimination of tail recursion and tail recursion mod constructors) with our bottom-up implementation for structural recursive functions.

Structural recursion arises naturally with inductively defined data types. Suppose we have a finite set Σ of data constructors with arities. Let T_Σ denote the set of all ground terms over Σ.

Definition. A function f defined by $f(x_1, \ldots, x_n) = e$ is *structural recursive defined* (srd) if there is a *recursion argument* $x_r \in T_\Sigma$ and inside of e there is a case distinction **case** x_r **of** $\ldots \sigma_j(z_1, \ldots, z_k) : e_j \ldots$ on x_r's top symbol (*e.g.* σ_j) in such a way that all recursive calls of f occur inside of some e_j and the rth argument of f in a recursive call is one of $z_1, \ldots, z_k$, *i.e.*, an immediate subterm of x_r.

The definition (as well as the evaluation scheme described below) can be generalized to mutual structural recursive functions and to functions with finite course-of-value recursion. See [3] for more information.

2 Bottom-up Evaluation

Evaluation of srd functions is done with visits to the nodes of the recursion argument. Visits are a well-known concept from evaluators for attribute grammars. A traversal of the derivation tree of the underlying grammar is a sequence of visits to its nodes. In our approach the recursion argument plays the rôle of the derivation tree. The tree traversal is compiled to iterative code. It works by reversing the tree pointers (similar to the Schorr-Waite garbage collection algorithm) and by saving a continuation address in the tag field for the identification of the data constructor. The information in the tag field can be recovered after the visit since it is implicit in the continuation address. Thus the state of the traversal (the node which is just visited, what action to perform next, the way back to the root of the tree) can be captured in two registers and in some part of the recursion argument. The first approach to bottom-up evaluation — an interpretative approach without continuation addresses — is found in [1], a condensed description of our current technique is [4], more details and an introduction are found in [2].

3 Structure of the system

LaToKi is written in Edinburgh SML. It consists of three parts. The first is a front end covering syntax analysis and language dependent semantic analysis. Currently, there is a front end for ModAs/6000 (developed from the language of [1]) and for an experimental language whose only data type is tree. Front ends are planned for a functional subset of SML and for a language based on order-sorted algebra.

The second part works on a common abstract syntax. It does language independent analyses and compiles to the abstract iterative machine AIM (cf. [4]). Here, the compiler examines each function definition and chooses the appropriate implementation strategy. We thank Jochen Spranger for its implementation.

The third part is a native code generator. Currently, we have a prototype code generator for the RS/6000. An optimizing version of it is under development. We also plan a version for SPARC processors. Recently, many groups are compiling functional languages into C. This was not possible here, since we need explicit access to return addresses for efficiency reasons.

4 Performance

Each of the implementation strategies mentioned above has its advantages. The current code generator only implements bottom-up evaluation (srd) and the usual runtime stack method (rec). In the table below we give some measurements for sample code (without optimization and garbage collection). We have applied the functions **append** and **nreverse** to lists of various lengths with both evaluation techniques.

nreverse	elapsed time		user time	
length	rec	srd	rec	srd
1000	1.1	0.9	1.10	0.95
2000	4.6	3.8	4.50	3.63

append	elapsed time		user time	
length	rec	srd	rec	srd
10^5	0.3	0.2	0.29	0.23
10^6	84.8	2.3	5.71	2.31

References

1. Herbert Klaeren and Klaus Indermark. Efficient implementation of an algebraic specification language. In M. Wirsing and J. A. Bergstra, editors, *Algebraic Methods: Theory, Tools, and Applications*, pages 69–90. Springer, 1987. LNCS 394.
2. Peter Thiemann. Efficient implementation of structural recursive programs. Technical Report WSI-91-12, Universität Tübingen, 1991.
3. Peter Thiemann. Konzepte zur effizienten Implementierung strukturell rekursiver Programme. Dissertation, Fakultät für Informatik, Universität Tübingen, 1991.
4. Peter Thiemann. Optimizing structural recursion in functional programs. In *Proceedings International Conference on Computer Languages 1992*, pages 76–85, April 1992.

This article was processed using the LaTeX macro package with LLNCS style