A Target Implementation for High-Performance Functional Programs

Sergio Antoy and Andy Jost

Computer Science Dept., Portland State University, Oregon, U.S.A.
antoy@cs.pdx.edu
ajost@synopsys.com

Abstract. We present a target implementation of a class of functional logic programming languages. We benchmark the functional component of our implementation on a small set of simple programs and compare its performance against the Glasgow Haskell Compiler. The results indicate that our approach is competitively efficient. We briefly outline the key characteristics of our implementation: underlying theory, data layout, memory management, execution, and optimizations.

1 Introduction

Our goal is a high-performance implementation of functional logic programs, suitable for use as the target of languages such as Curry [11] or $\mathcal{TOY}$ [7]. The evaluation strategy for such languages is the subject of ongoing research, and, within this area of inquiry, one might ask what a suitable starting point for implementing and testing new ideas might be. A favorite approach has been to map the functional logic program to an existing target language in either the functional [6] or logic [12] domains. This is a reasonable approach, given the difficulty involved in implementing any programming language, but the solution is not without its own limitations. For one, it only solves half of the problem. If the target language provides native support for, say, functional constructs, then the logic constructs are still left up to the developer (and vice versa). Another limitation is that it allows details of the underlying language (or its implementation) to appear in awkward places and ways. For instance, a functional logic program expressed in Prolog might be inclined to use a backtracking strategy to handle non-determinism; but this is a major inconvenience when the strategy for implementing non-determinism is itself in question! Likewise, an implementation in Haskell may simulate non-determinism by wrapping every type in the target program, but this approach may add significant overhead [6].

We believe there is a better way. We aim to support *both sides* of the functional logic paradigm in a unified *target implementation* that serves as the target of a language compiler. It should be small and understandable, and highly efficient so that it serves as a suitable platform for experimentation outside the bounds of the traditional functional and logic paradigms. Functional logic computations are executed by steps classified as either deterministic (abstracting the functional component of the paradigm) or non-deterministic (abstracting the logic component). The majority of the steps of the computation of a typical program are deterministic. This contribution describes our implementation of these steps. Non-deterministic steps are conceptually very similar except for managing the non-deterministic alternatives of a choice. The techniques employed to this aim, e.g., backtracking or pull-tabbing [3], are largely orthogonal to the implementation of the steps, and can be selected and added later. In this way, we may say how our approach compares with existing implementations of deterministic computations before moving into difficult territory.

2 Benchmark Results

We have coded a prototype demonstrating our target implementation for functional computations. The implementation, code-named *Sprite*, written in C++, has been tested on x86_64 hardware running Linux. Our initial results versus the Glasgow Haskell Compiler (*ghc*) [9] are shown in Fig. 1. *ghc* was run with the `-O2` flag to enable optimizations. Of the five benchmarks, `rfib` and `tak` come from both the Haskell `NoFib` [14] and the KiCS2 [10] benchmark suites, `exp3_8` from `NoFib` only, `reverse` from KiCS2 only, and `tree_insert` from ViaLOIS [4]. In some cases the input size has been increased when the program would otherwise finish too quickly (the Linux `time` command, which we used to collect these data, has a resolution of 10 ms).

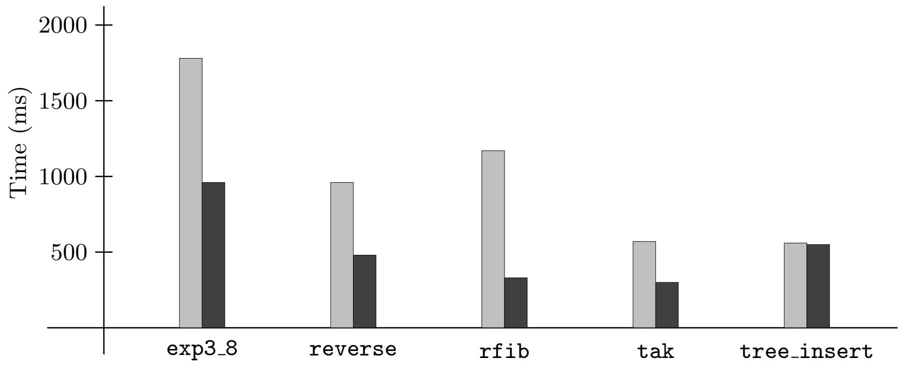

Fig. 1. Benchmark results. ▨ *ghc*, ■ *Sprite*. Times are in milliseconds.

The salient features of our target implementation and its prototype are detailed below in terms of a very simple generic computer system in which we assume the existence of a few low-level abstractions that are common to most systems: a native word size that is subdividable and large enough to hold a pointer to any region of memory; a set of global word-sized registers; and an operating system that provides basic services, such as for dynamic memory management. Our aim is to describe the target implementation in a way that can be more-or-less directly mapped to many modern computer systems without committing to a particular system in the description itself.

Sprite is a prototype of this implementation approach targeted to a specific computer system consisting of 1) an implementation language (C++), 2) a hardware architecture (x86_64), 3) an application binary interface (AMD64), and 4) an operating system (Linux). Although *Sprite* is realized in terms of these particulars, that fact should in no way detract from the generality of our approach, much as, say, an implementation of Linux for AMD64 does not take away from the general usability of Linux on other systems.

3 Target Details

Both theory and practice of the implementation of functional logic languages are rather specialized. In this section we can only highlight some distinctive features of our work.

3.1 Underlying Theory

A program is seen as an *inductively sequential graph rewriting system* [8]. A *node* of a graph (also called an *expression*) abstracts a *label* (a symbol of the program signature) and a sequence of *successors* (the arguments to which the symbol is applied). A *computation* is a sequence of rewrite steps. A *step* replaces a subexpression (also called the *redex*) of the expression being evaluated with a new expression (called the *replacement*). A symbol is abstracted by a structure, described shortly, that stores data, e.g., a node's successors, and virtual methods, e.g., for computing a step.

3.2 Data Layout

Nodes are represented in a four-word structure comprising a virtual table pointer (*vptr*), one word of metadata, and two word-sized slots. The *vptr* enables the familiar mechanism for dispatching type-based actions at run time. The metadata field is split into two parts. The first part contains a signed integer called *tag*, which indicates the node type. The type distinguishes between *defined operations* and *data constructors* of the functional logic program, and a few other possibilities, such as *failures* and *non-deterministic choices*, which are not used in the work reported in this paper. The second part holds the mark used by the garbage-collection algorithm.

The pair of slots is used to hold either a data payload or successors. A node that represents a built-in value (e.g., an integer) places its payload in one or both slots. If the payload is larger than two words, then the first data slot instead contains a pointer to the payload. When a node has successors, they are stored in the slots, the exact layout depending on the arity. For nodes with fewer than three successors, pointers to the successors are stored directly in the slots.

Otherwise, one slot holds a pointer to a dynamically-allocated region of memory large enough to contain all of the successors.

A rewrite step overwrites a redex in place (destructively) by use of the C++ *placement new* operator. The rewrite step can and often will change the dynamic type[1] of the redex. In other words, virtual methods invoked at the redex before and after rewriting may resolve differently. The importance of this will be clear shortly. For rewriting to work correctly, every node must fit within the same allocation size — the four-word layout described above.

Memory Management. We assume the computer system provides nothing more in the way of memory management than primitive operations to dynamically allocate and de-allocate memory. A simple mark-and-sweep garbage collector manages allocated memory. Memory pools are used to improve the efficiency of memory allocations. The basic pool implementation is provided by the *Boost.Pool* library [5]. In this implementation, each pool contains zero or more blocks that provide storage for many objects, and has a single variable representing the head of a free list. Allocation from one of these pools usually requires little work. On x86_64 machines, in the common case where the pool is not exhausted, it requires two instructions (a test and a branch) to check the head of the free list, one instruction to copy the current head (the allocated node address), and one more instruction to advance the head of the free list. When the pool is exhausted, a new block is allocated, roughly doubling the total allocation size.

3.3 Rewriting

A rewrite step of an expression e is computed by dispatching a virtual function, called *step*, on the root node of e. For each defined operation symbol, f, function *step* is generated by a traversal of the *definitional tree* of f [1, 2] roughly as described in [4, Fig. 1]. For every data constructor, function *step* performs no operation.

The execution of function *step* for an expression e rooted by an operation symbol either finds a step of e or it aborts the computation if no step is found (and consequently e has no value). Function *step* performs pattern matching and determines whether expression e is a redex. If it is, it computes its replacement and rewrites e. If it is not, it recursively invokes *step* for the root of some subexpression of e. In any case, the step computed by *step* is *needed* [13] to obtain the value of e. Hence, the evaluation strategy is theoretically as efficient as it could be.

[1] Where it is convenient to do so, we will speak in terms of the *dynamic type* of a node and its *virtual methods*. By analogy with the most common implementations of C++, this type is determined entirely by the contents of the vptr, and virtual method invocations are type-dependent calls dispatched through the *vptr*. It is assumed the reader is familiar with the terminology. Although this mechanism strictly does not exist in a generic computer system, it can easily be emulated, and the analogy usefully simplifies our description of rewriting.

3.4 Performance Optimizations

Our approach is compatible with and benefits from both generic and specific optimizations. Generic optimizations include, e.g., deforestation [16] and tail calls [15]. These are independent of our approach and we are mostly ignoring them in our benchmarks. We replace a tail recursive call with a jump only in one function of *tree_insert*.

The specific optimizations originate from the underlying theory of our approach. The target code reduces only *needed* redexes, i.e., redexes of an expression e that must be reduced in *every* derivation of e to a value. Furthermore, these redexes not only must be reduced, but also must be derived to constructor-rooted expressions. The target code takes advantage of this condition to re-use from some step to the next one some comparisons for pattern matching, and to avoid the creation of some operation-rooted expressions that would immediately be taken apart and discarded as garbage.

Furthermore, for some types such as the integers, a constructor-rooted expressions is a literal value. When an integer expression is *needed*, it can be evaluated in an eager-like fashion and passed from producers to consumers unboxed. These conditions explain the superior performance of our approach in benchmarks such as *rfib* and *tak*.

3.5 Machine Register Usage

Sprite reserves four general-purpose machine registers for its own exclusive use. Since a target program necessarily devotes a large fraction of its issued instructions to pattern matching and rewriting, a few frequently-referenced variables related to those tasks, such as the address of the current redex and head of the free list used to allocate node storage, are stored in the dedicated registers. This technique improves performance by eliminating any address computations for those variables, and also work related to passing them from one function to the next, or moving their values between the processor and memory. Our measurements suggest the cost — that fewer general-purpose registers are available to the program — is easily outweighed by these benefits.

3.6 Program Encoding

Source programs are encoded in C++ with the help of a macro language. Data constructors are specified as their label and arity, data types as an ordered sequence of constructors. Defined operations are expressed with a recursive structure that closely matches the structure of a definitional tree. Ignoring the optimizations specifically mentioned in Sect. 3.4, the source program has an unambiguous[2] translation into the macro language. Though currenly done by hand, the encoding process is highly mechanical and, therefore, does not permit one to

[2] Aside from name-mangling and name-spacing conventions, of which there are many suitable approaches.

commit abuses to gain performance. The manual approach limits us to a small number of simple programs, but is adequate for rapidly exploring the implementation space.

4 Conclusions

We have described the design of a target implementation for functional programs. We have presented details about a prototype implementation of this design, called *Sprite*, that efficiently performs functional computations. The preliminary performance results suggest *Sprite* may in some cases achieve higher performance than *ghc*. Inferring definitive conclusions from a benchmark of five programs would be hazardous at best. Furthermore, our benchmark is biased toward programs that execute only a small number of simple functions.

In our benchmark, it is desirable to compare the execution of the same source program by ghc and Sprite. Ghc might optimize a program by some source-to-source transformations. Sprite can accommodate and benefit from the same transformations, but since we do not know whether ghc optimizes a program in such a way, we prefer small, simple programs in which these transformations are less likely to occur and/or can be more easily predicted.

Also, we compile our programs largely by hand. This compilation is tedious, error-prone, and does not inspire confidence. Small, simple programs are easier to compile in this way and are less likely to contain plausible optimizations that a compiler may not be able to infer. We have been careful and conservative, but without an actual compiler our conclusions must be considered preliminary. The implementation of such a compiler is our next goal.

References

1. S. Antoy. Definitional trees. In H. Kirchner and G. Levi, editors, *Proceedings of the Third International Conference on Algebraic and Logic Programming*, pages 143–157, Volterra, Italy, September 1992. Springer LNCS 632.
2. S. Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 40(1):875–903, 2005.
3. S. Antoy. On the correctness of pull-tabbing. *TPLP*, 11(4-5):713–730, 2011.
4. S. Antoy and A. Peters. Compiling a functional logic language: The basic scheme. In *Proceedings of the Eleventh International Symposium on Functional and Logic Programming*, pages 17–31, Kobe, Japan, May 2012. Springer LNCS 7294.
5. *Boost C++ Libraries*, 2013. Available at `http://www.boost.org/`.
6. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A new compiler from Curry to Haskell. In *Proceedings of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.
7. R. Caballero and J. Sánchez, editors. *TOY: A Multiparadigm Declarative Language (version 2.3.1)*, 2007. Available at `http://toy.sourceforge.net`.
8. R. Echahed and J. C. Janodet. Admissible graph rewriting and narrowing. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 325 – 340, Manchester, June 1998. MIT Press.

9. *The Glasgow Haskell Compiler*, 2013. Available at http://www.haskell.org/ghc/.
10. M. Hanus. KiCS2 benchmarks. Available at http://www-ps.informatik.uni-kiel.de/kics2/benchmarks/, 2011.
11. M. Hanus, editor. *Curry: An Integrated Functional Logic Language (Vers. 0.8.3)*, 2012. Available at http://www-ps.informatik.uni-kiel.de/currywiki/.
12. M. Hanus, editor. *PAKCS 1.11.2: The Portland Aachen Kiel Curry System*, 2013. Available at http://www.informatik.uni-kiel.de/~pakcs.
13. G. Huet and J.-J. Lévy. Computations in orthogonal term rewriting systems. In J.-L. Lassez and G. Plotkin, editors, *Computational logic: essays in honour of Alan Robinson*. MIT Press, Cambridge, MA, 1991.
14. W. Partain. The NoFib benchmark suite of Haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, London, UK, 1993. Springer-Verlag.
15. G. Steele, Jr. Debunking the "expensive procedure call" myth or, procedure call implementations considered harmful or, lambda: The ultimate goto. In *Proceedings of the 1977 ACM annual conference*, pages 153–162, New York, NY, USA, 1977.
16. P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231–248, 1990.