©Springer-Verlag
In *Proc. of the 16th International Workshop on Implementation and Application of Functional Languages, IFL 2004*, Lübeck.
Springer LNCS 3474, pp. 108–125, 2005

A Virtual Machine for Functional Logic Computations[⋆]

Sergio Antoy[1], Michael Hanus[2], Jimeng Liu[1], and Andrew Tolmach[1]

[1] Portland State University, Computer Science Dept.
P.O. Box 751, Portland, OR 97207, U.S.A.
`{antoy,jimeng,apt}@cs.pdx.edu`

[2] Christian-Albrechts-Universität Kiel, Institut für Informatik
Olshausenstr. 40, D-24098 Kiel, Germany.
`mh@informatik.uni-kiel.de`

Abstract. We describe the architecture of a virtual machine for executing functional logic programming languages. A distinguishing feature of our machine is that it preserves the operational completeness of non-deterministic programs by concurrently executing a pool of independent computations. Each computation executes only root-needed sequential narrowing steps. We describe the machine's architecture and instruction set, and show how to compile overlapping inductively sequential programs to sequences of machine instructions. The machine has been implemented in Java and in Standard ML.

1 Introduction

Functional logic programming aims at integrating the characteristic features of functional and logic programming into a single paradigm. In the last decade, the theory of functional logic computations has made substantial progress. Significant milestones include a model that integrates narrowing and residuation [13], narrowing strategies for several classes of programs suitable for functional logic languages [5], a functional-like model for non-deterministic computations [3], and well-defined semantics for programming languages of this kind [1, 11].

These results have been influential in the design and implementations of functional logic programming languages, e.g., Curry [18] and $\mathcal{TOY}$ [19]. Most existing implementations of these languages are based on a translation of source code to Prolog code (e.g., [7]), which can be executed by existing standard Prolog engines. This approach simplifies the task of implementing functional logic language features: e.g., source language variables can be implemented by Prolog variables and narrowing can be simulated by resolution. But some problems arise; most notably, the depth-first evaluation strategy of the Prolog system causes the loss of the operational completeness of functional logic computations and inhibits the implementation of advanced search strategies [17].

[⋆] This work was supported in part by the National Science Foundation under grants CCR-0110496 and CCR-0218224 and by the German Research Council (DFG) under grants Ha 2457/1-2 and Ha 2457/5-1.

This paper describes a fundamentally different approach to the implementation of a functional logic language, namely a virtual machine for functional logic computations. Section 2 sketches the key features of functional logic languages. Section 3 describes the architecture of the virtual machine. In particular, we describe how functional logic features influence several key decisions, e.g., non-determinism and the desire for operational completeness suggest an architecture that executes a pool of independent computations concurrently. We describe the kind of steps executed by each computation in the pool. By choosing a specific class of source programs, we can arrange that the machine only needs to execute root-needed steps sequentially, a characteristic that promotes both simplicity and efficiency. We describe the registers of the machine, the information they contain, and how the machine instructions control the flow of information between these registers. Finally, we sketch how a program can be compiled into machine instructions. Examples are provided throughout the discussion. Section 4 describes on-going efforts at implementing the virtual machine in both Java and Standard ML. The Java implementation, which is the more highly developed, is mainly intended as a compiler/interpreter for Curry, but it could be used to interpret compiled functional logic programs coded in other languages. Section 5 contains the conclusion and a brief discussion of related work.

2 Functional Logic Computations

Functional logic computations generalize functional computations by adding three specific features: non-determinism, narrowing and residuation (see [12] for a survey). Our machine is not designed for a specific programming language. The examples in this paper are in Curry, but the details of the source language are largely irrelevant. Our only assumption is that source programs can be converted to a particular variety of first-order term rewriting systems. The requirements on these rewriting systems are described in more detail below.

2.1 Functional Logic Features

Non-determinism is the feature that allows an expression to have multiple distinct values. Non-determinism broadens the class of programs that can be coded using functional composition [3]. For example, a program that solves a *cryptarithm* must assign digits to each letters. This can be expressed as "`let s = digit in...`" where `digit` is defined by the rules

```
digit = 0
digit = 1
...                                                    (1)
digit = 9
```

The rules of `digit` are *not* mutually exclusive, i.e., the expression `digit` has 10 distinct values. The value eventually chosen for a given letter is constrained, according to a cryptarithm, by some other part of the program. All the rewrite rules of function `digit` have the same left-hand side. (In Sections 3.6 and 3.7, we will consider these 10 rules as a single rule where the right-hand side is non-deterministically chosen among 10 possibilities. A justification of this viewpoint and the opportunity to exploit it for an efficient evaluation strategy are in [3].)

Narrowing is the glue between functional and logic computations. The execution of a functional logic program may lead to the evaluation of an expression containing an uninstantiated variable. Narrowing "guesses" a value for the variable when this is necessary to keep the computation going. For example, the function that returns the last element of a list can be coded as follows ("++" is the list concatenation function):

```
last l | l =:= x++[e] = e  where x,e free                (2)
```

The evaluation of `last [1,2,3]` prompts the evaluation of `[1,2,3] =:= x++[e]`, the rule's condition (e_1 `=:=` e_2 denotes the equality constraint that is satisfied if e_1 and e_2 are evaluable to unifiable data terms). The variables `x` and `e` are uninstantiated. Narrowing finds values for these variables that satisfy the condition; this is all it takes to compute the last element of the input list.

Residuation is an alternative mechanism for handling evaluation of an expression containing an uninstantiated variable. In this case, the evaluation suspends, and control is transferred to the evaluation of another expression in hopes that the latter will instantiate the variable so that the former can resume execution. (Evidently this only makes sense when more than one subexpression is available to be evaluated, e.g., the conjuncts of a "parallel and" operation.) The decision of whether to narrow or residuate is specified by the programmer on a per-function basis. Generally, primitive arithmetic operations and I/O functions residuate, since it seems impractical to guess values in these cases, whereas most other functions narrow.

2.2 Overlapping Inductively Sequential Rewrite Systems

Our abstract machine is intended to evaluate programs that can be expressed as *overlapping inductively sequential term rewriting systems* [3]. Roughly speaking, this means that pattern matching can be represented by (nested) case expressions with multiple right-hand sides for a single pattern. More precisely, every function of an overlapping inductively sequential system can be represented by a particular variety of *definitional tree* [2, 3], which we specify in Section 3.7.

It is shown in [4] that every functional logic program defined by constructor-based rewrite rules, including programs in the functional logic languages Curry and $\mathcal{TOY}$, can be transformed into an overlapping inductively sequential system. This class properly includes the first-order programs of the functional languages ML and Haskell. Higher-order features, i.e., applications of a functional expression to an argument, can be represented as an application of a specific first-order function *apply* (where partial applications are considered as data terms)—a standard technique to extend first-order languages with higher-order features [23]. (Additional preliminary compiler transformations, e.g., name resolution, lambda lifting, etc., are typically needed to turn source programs into rewrite system form; we do not discuss these further here.)

3 Virtual Machine

In this section we describe how the features of functional logic computations, in particular non-determinism and narrowing, shape the architecture of our virtual machine. We only sketch the machine's support for residuation; full details of this are beyond the scope of this paper.

3.1 Pool of Computations

A fundamental aspect of functional logic computations is non-determinism—both in its ordinary form, as in example (1), and through narrowing, as in example (2). The execution of a non-deterministic step involves one of several choices in the replacement of a redex—or, to use a more appropriate term in our environment, a *narrex*. (In the remainder of the paper, we use "narrowing" to refer to either narrowing or rewriting, which is a special case of narrowing.) For example, in the cryptarithm solver mentioned earlier, the evaluation of `digit` leads to 10 possible replacements.

One of our main goals is to ensure the operational completeness of computations. For instance, consider the following function to reverse the elements in a list:

```
rev (x:xs) = rev xs ++ [x]
rev []     = []
```
(3)

A complete computation mechanism will be able to compute a solution to the equation `rev l =:= [1,2]`, namely {`l=[2,1]`}. A conventional backtracking policy that tries each clause of `rev` in order will loop forever on the first clause, and hence is not complete. The simplest policy to ensure completeness is to execute any non-deterministic choice fairly, independently of the other choices. In our virtual machine, this is achieved by concurrently computing the outcome of each replacement. In our machine, a *computation* is explicitly represented by a data structure, which holds the term being evaluated, a substitution, and a state indicator with values such as *active*, *complete*, or *residuating*.

The machine maintains a *pool* of computations. Initially, there is only one active computation in the pool, containing the initial *base term*. Computations change state depending on events or conditions resulting from the execution of machine instructions. For example, when a computation makes a non-deterministic step, the computation is *abandoned*; new computations, one for each possible step, are created, added to the pool, and become *active*. When a computation obtains a normal form or a head normal form (we have a different kind of computation for each task), the computation state is set to *complete*.

The core of the machine is an engine to perform head normal form computations, by executing sequences of machine instructions. There is one such sequence associated with each function of the source program, which we call the *code* of the function. The purpose of a function's code is to perform a narrowing step of an application of the function to a set of arguments, or to create the conditions that lead to a narrowing step (details are given in Section 3.3). The instructions operate on an internal *context* consisting of several registers and stacks (described in Section 3.5). The instruction sequence is always statically bounded in length, and contains no loops. For the simplest functions, it is just a few instructions long. For more complicated functions, the number of instructions goes up to a few dozen, but seldom more than that. When the virtual machine completes the execution of a function's code, most of the context information become irrelevant.

To manage the pool of computations fairly, the machine must share the processor among active computations so that they make some "progress" toward a result over time. We considered several strategies to ensure fair sharing. For example, a fixed amount of time could be allocated to each computation. If a computation C ends before the expiration of its time, a different computation is executed. Otherwise, C is interrupted. When all the other computations existing in the pool at the time of the interruption of C have received their fair share of time, the execution of C resumes. A similar strategy would be to allocate a fixed number of virtual machine instructions.

A drawback of the above strategies is that when a computation is interrupted, the instruction execution context must be saved, and subsequently restored when the computation resumes. In order to minimize the overhead of switching contexts, we have adopted a simpler strategy that never interrupts instruction sequences. This remains fair because the length of each instruction sequence is bounded. When the machine selects a computation from the pool, it executes the entire code of some function for that computation, and then returns the computation to the pool. It then repeats this process fairly for every other computation of the pool.

3.2 Terms and Computations

In the model for functional logic programming described in [13], a computation is the process of evaluating an expression by narrowing. The expression is a term of the rewrite system modeling the program. A *term* t is a *variable* v or a *symbol* s of fixed arity $n \geqslant 0$ applied to m terms $t_1, \ldots, t_m$, $m \leq n$, written as $s(t_1, \ldots, t_m)$. Symbols are partitioned into data *constructors* c and *functions* or *operations* f. A *data term* is a term without defined functions, a *pattern* is a function applied to data terms, and a *head normal form* is a term without a defined function at the root, i.e., a variable or a constructor-rooted term. In examples, we often write terms using infix notation for symbols. A position *pos* in a term is represented by a sequence of positive integers representing subterm choices, beginning at the root. For example, the position of `x` in `f(y,b(x,z))` is the sequence $2{\cdot}1$. We write $t|_{pos}$ for the subterm at position *pos* in t.

Evaluating a term results in both a *computed value*, as in functional programming, and a *computed answer*, as in logic programming. The computed value is a data term, and the computed answer is a substitution, possibly the identity, from some free variables of the term being evaluated to data terms. In Example (2), the evaluation of `[1,2,3] =:= x++[e]` returns the computed value `Success`, a predefined constant for constraints, and the computed answer $\{$`x` $\mapsto$ `[1,2]`, `e` $\mapsto$ `3`$\}$.

Thus, the state of a computation includes both a term and a substitution. Initially, the computation data structure for a term t holds t itself and the identity substitution. As narrowing steps are executed, both the term and the substitution fields of the computation structure are updated. A computation is *complete* when the machine cannot perform a step in the term being evaluated.

The machine supports three kinds of computations. **Normal form computations** attempt to narrow terms all the way to data terms. The virtual machine is intended to be used within a host program that provides the read-eval-print loop typical of many functional and logic interpreters. The host program provides the initial base term for the machine to evaluate to normal form, and waits for the computed values and answers to be returned (if the program narrows variables or executes non-deterministic steps, multiple value/answer results are possible).

Head normal form computations try to evaluate terms to constructor-rooted terms or variables. Executing these computations is the core activity of the machine, during which the definitions of functions are applied. Since normal form computations can be modeled by head normal form computations using auxiliary operations (see, e.g., [15]), we concentrate on head normal form computations in this paper; they are described in more detail in Section 3.3.

Parallel-and computations handle the evaluation of a conjunction of two terms. Residuation is only meaningful in the presence of these computations. Each conjunct is evaluated by a different computation. For each conjunction, the computation of one and only one of the two conjuncts is active at any one time (implementing an interleaving semantics for concurrency [13]). If the computation of the first conjunct residuates, the computation of the second one becomes active. The second computation may "unblock" the first one, thus becoming *waiting* itself, or may residuate as well. In this case, the entire computation blocks. If all the parallel-and computations derived from a given base term are blocked, the base term computation *flounders*. Since we are omitting details of residuation support in this paper, we ignore parallel-and computations in subsequent sections.

The computations in the machine's pool are conceptually independent of each other. In our implementation, the evaluation of some subterms common to two independent computations may be shared, but this is only for the sake of efficiency. Thus, we describe the execution of a computation disregarding the fact that other computations may be present in the pool.

3.3 Head Normal Form Computations

The execution of a head normal form computation attempts to rewrite an *operation*-rooted term into a *constructor*-rooted term or variable. The evaluation strategy executed by our machine is *root-needed* reduction [21] with the addition of narrowing and non-deterministic steps. Simply put, the strategy repeatedly attempts to apply rewrite rules at the top of an operation-rooted term until a constructor-rooted term or variable is obtained.

The implementation of this strategy for a given function depends only on the forms of the left-hand sides of that function's defining rules. In fact, the definitional trees that our system uses to represent programs already implicitly encode the strategy. The next needed step in the evaluation of a term $f(t_1, \ldots, t_n)$ can be obtained by comparing the symbols at certain positions in the arguments of f with corresponding symbols in f's definitional tree. A sequence of comparisons determines which rule to apply, or which subterm to evaluate. To implement these tree-based operations, we compile the definitional tree for each function f to a code sequence of virtual machine instructions, as described in Section 3.7. The instructions themselves are described in Section 3.6.

The code for a function effectively chooses which rule to apply to a term. But it is also possible that *no* rule can be applied at the top of an operation-rooted term. This can occur for one of only two reasons: (1) an operation-rooted argument of a function application must be evaluated to a head normal form before any rule can be applied, or (2) the function is incompletely defined. An example of each condition follows. Consider the definitions of the usual functions that compute the head of a list and the concatenation of lists, denoted by the infix operator "++".

```
head (x:_) = x

[]     ++ y = y                                              (4)
(x:xs) ++ y = x : xs ++ y
```

The term $t = \texttt{head}\,(u \mathbin{+\!+} v)$, for any u and v, is an example of the first condition. To evaluate t, it is necessary to evaluate $(u \mathbin{+\!+} v)$ which is a recursive instance of the original problem, i.e., to evaluate an operation-rooted term to a head normal form.

The term $t = \texttt{head []}$ is an example of the second condition. In a deterministic language, where the execution of a program consists of a single computation, this condition is usually treated as an error. In a non-deterministic language, where the execution of a program may consist of several independent computations, this condition is often benign. The machine uses a distinguished symbol, which we denote by `fail`, to replace terms that have no value. Since for every computation of the pool the machine executes exclusively needed steps, the reduction of any subterm to `fail` implies that the entire computation should fail.

3.4 Data Representation

We now describe the virtual machine more formally. The terms manipulated by the machine are represented by acyclic directed graphs stored in heaps. This graph-based representation of terms is necessary to capture the intended sharing semantics of the language, and also allows us to express important optimizations when manipulating and replacing subterms. Formally, a *heap* is a finite map $\Gamma : H \rightarrow P + V$, where H is an abstract set of *handles* (e.g., heap addresses), P is a set of pairs of the form $\langle s, (h_1, \ldots, h_n)\rangle$, where s is a program symbol of arity $m \geqslant 0$, $n \leqslant m$, and $h_1, \ldots, h_n$ are handles, and V is a set of program variables v. (We distinguish elements of P from those of V by always writing the former using pair notation.) The term *represented* by handle h in heap Γ is given by

$$trm_\Gamma(h) = \begin{cases} s(trm_\Gamma(h_1), \ldots, trm_\Gamma(h_n)) & \textit{if } \Gamma(h) = \langle s, (h_1, \ldots, h_n)\rangle \\ v & \textit{if } \Gamma(h) = v \end{cases}$$

We make extensive use of finite maps in what follows, so we fix some general notation for these here. If M is a finite map, then $M[u := v]$ is the result of extending or updating M with a mapping from u to v. We write $\emptyset$ for an empty map, and $[u := v]$ as shorthand for the singleton map $\emptyset[u := v]$. We write $Dom(M)$ for the domain of M.

The storage areas of the machine (described in Section 3.5) hold handles for terms; more loosely, we sometimes just say they hold terms and we extend some standard term rewriting notations to handles. For example, if h is handle and $p = p_1 \cdot p_2 \cdots p_n$ is a position, then we define

$$h|_{p_1 \cdots p_n} = h_{p_1}|_{p_2 \cdots p_n} \textit{where } \mathit{\Gamma}(h) = \langle \_, (h_1, \ldots, h_n) \rangle$$

It follows immediately that $trm_\mathit{\Gamma}(h|_p)$ = $trm_\mathit{\Gamma}(h)|_p$. We also define the set of *subhandles* of a handle in the obvious way:

$$\mathit{shs}_\mathit{\Gamma}(h) = \begin{cases} \{h\} \cup shs_\mathit{\Gamma}(h_1) \cup \ldots \cup shs_\mathit{\Gamma}(h_n) & \textit{if } \mathit{\Gamma}(h) = \langle s, (h_1, \ldots, h_n) \rangle \\ \{h\} & \textit{if } \mathit{\Gamma}(h) = v \end{cases}$$

For any handle h, the terms represented by the handles in $\mathit{shs}_\mathit{\Gamma}(h)$ are just the subterms of $trm_\mathit{\Gamma}(h)$.

Substitutions σ are finite maps from handles to handles, where the handles of the domain typically (but not necessarily) represent variables. Substitutions are never applied destructively to change a term in-place, since different computations might need to apply different substitutions to a same term. Instead, they are applied to handles representing terms by making a clone (deep copy) of the term. More precisely, we define a "clone with substitution" operator as follows:

$$\mathit{clone}_\sigma(\mathit{\Gamma}_0, h) = \begin{cases} (\mathit{\Gamma}_0, \sigma(h)) & \textit{if } h \in Dom(\sigma) \\ (\mathit{\Gamma}_0, h) & \textit{if } \mathit{shs}_\mathit{\Gamma}(h) \cap Dom(\sigma) = \emptyset \\ (\mathit{\Gamma}', h') & \textit{otherwise, where} \\ & \mathit{\Gamma}_0(h) = \langle s, (h_1, \ldots, h_n) \rangle \\ & (\mathit{\Gamma}_i, h'_i) = \mathit{clone}_\sigma(\mathit{\Gamma}_{i-1}, h_i) \quad (1 \leq i \leq n) \\ & \mathit{\Gamma}' = \mathit{\Gamma}_n[h' := \langle s, (h'_1, \ldots, h'_n) \rangle] \quad (h' \notin Dom(\mathit{\Gamma}_n)) \end{cases}$$

This *clone* operator is quite efficient since it copies (only) the spines of the term above any substituted variables; any parts of the source term remaining unaffected by the substitution are shared by the result term. For cloning to have the expected substitution semantics on the represented terms, it is important that no variable appears more than once in the heap; i.e., if $trm_\mathit{\Gamma}(h_1) = v$ and $trm_\mathit{\Gamma}(h_2) = v$, then $h_1 = h_2$. We call heaps having this property *well-formed*, and we take care to start the machine with a well-formed heap and maintain the well-formedness invariant during execution. Suppose $\mathit{\Gamma}$ is well-formed, $trm_\mathit{\Gamma}(h) = t$ and $trm_\mathit{\Gamma}(j) = u$ for some terms t and u, and $trm_\mathit{\Gamma}(k) = v$ for some variable v. If $(\mathit{\Gamma}', h') = \mathit{clone}_{[k:=j]}(\mathit{\Gamma}, h)$, then $trm_{\mathit{\Gamma}'}(h') = t[u/v]$, i.e., the usual term substitution of u for v in t.

3.5 Storage Areas

As discussed in the previous sections, our machine fairly executes a pool of independent computations. The context of each computation includes a heap and four separate *storage areas*, a generic name for stacks and registers.

Suppose that t is the term to evaluate in a head normal form computation. We recall that initially t is operation-rooted; the computation completes successfully when t is evaluated to a constructor-rooted term or variable. The computation begins by executing the code associated with the function at the root of t. In the course of executing this code, it may become necessary to recursively evaluate operation-rooted subterms of t. The **pre-narrex stack** keeps track of these recursive computations. It is a stack containing handles $h_n, \ldots, h_2, h_1$ of a heap Γ, with h_n the top, having the following properties.

1. At the beginning of the computation, $n = 1$ and $trm_\Gamma(h_1) = t$.
2. Every term represented by a handle in the stack, with the possible exception of h_n, the top of the stack, is operation-rooted and it is not a narrex.
3. For all $i > 1$, h_i is a subhandle of h_{i-1} with the property that $trm_\Gamma(h_i)$ must be evaluated to a head normal form before $trm_\Gamma(h_{i-1})$ can be evaluated to a head normal form.

The top of the pre-narrex stack contains the term handle currently being evaluated. Referring to example (4), if `head` $(u \mathbin{++} v)$ is on the pre-narrex stack, then $u \mathbin{++} v$ will be pushed on the stack, too, because the former cannot be evaluated to a head normal form unless the latter is evaluated to a head normal form. The machine allocates a separate pre-narrex stack to each head normal form computation.

The other three storage areas are local to the execution of a single function code sequence.

Current register. This is a simple register containing a term handle. Many of the machine's instructions implicitly reference this register. For example, to apply a rewrite rule of the function "++" defined in (4) to the term $u \mathbin{++} v$, one must check whether the term u is rooted by `[]` or "`:`" or some function symbol. The BRANCH instruction that performs the test expects to find the term to be tested in the current register.

Pre-term stack. This is a stack for constructing narrex replacements. These are always term handles instantiating a right-hand side of a rule. The arguments of a symbol application are first pushed on the stack in reverse order. The MAKETERM instruction, which is parameterized by the symbol being applied, replaces these arguments with the application term. For example, the term `[1,2]++[3,4]`, which is a narrex, is replaced by `1:([2]++[3,4])` which is constructed as follows. First, the handles for the terms `[3,4]` and `[2]` are pushed on the pre-term stack. Executing "MAKETERM ++" replaces them with a handle to the new term `[2]++[3,4]`. Then, the handle for the term `1` is pushed on the stack as well and executing "MAKETERM `:`" replaces the two topmost elements with a handle for `1:([2]++[3,4])`.

Free variable registers. The rewrite rules that define the functions of the program can contain free (extra) variables. Several occurrences of a same free variable may be needed to construct the narrex replacement. Therefore, when a free variable is created, its handle is stored in a register (using instruction STOREVAR) to be retrieved later (using instruction MAKEVAR) if it occurs again. For example, consider the following rule that tells whether a string of odd length is a palindrome:

```
palind s  =  s =:= x ++ (y : reverse x)  where x,y free  (5)
```

The construction of an instance of the right-side of this rule begins with pushing `x`, for the right-most occurrence of the right-hand side, on the pre-term stack. Later on, another occurrence of x is to be pushed on the stack. Thus, a handle to x must be kept around so that it can be retrieved later and pushed again. The machine maintains the set of free variables as a finite map from variable index numbers (which are parameters to the STOREVAR and MAKEVAR instructions) to variable handles.

The content of these local storage areas can be discarded at the end of the execution of the function code. Since computations are never interrupted in the middle of an instruction sequence, there need only be one instance of these areas, which can be shared by all computations.

3.6 Machine Instructions

The virtual machine evaluates terms by executing sequences of instructions. Each instruction acts on the heap and the current computation to produces a (possibly) altered heap and zero or more new or changed computations. Thus, the behavior of a computation C in the current heap Γ will be specified as a transition $\Gamma, C \Longrightarrow \Gamma', \{C_1, \ldots, C_n\}$ ($n \geq 0$) where Γ' is a modified heap and $C_1, \ldots, C_n$ are the new or changed computations. Some instructions move information between the various storage areas. Others build or take apart terms. Building a term extends the heap; some other operations update it. Figure 1 gives transition rules for the instructions.

The machine begins a head normal form evaluation with a single active computation, containing a single term handle on the pre-narrex stack, and a well-formed heap. (The information in all the other storage areas is irrelevant.) The machine then repeats the following cycle. A computation is chosen (fairly) for execution from the active computation pool. If the top of the pre-narrex stack represents an operation-rooted term, the machine retrieves the code for the operation and begins to execute it. If the top of the pre-narrex stack represents a constructor-rooted term or a variable, the stack is simply popped; in this case an appropriate handle in the heap will already have been updated with that term. If the pre-narrex stack is empty, the computation is completed and is removed from the pool of active computations; the computed value can be read from the heap by the host program.

The code for a function is a sequence of instructions I. (In fact, because BRANCH instructions may contain multiple sub-sequences of instructions, the code really forms a tree.) The LOAD and BRANCH instructions deal with fetching and testing (handles of) existing terms. LOAD p extracts the subhandle at position p from the handle on top of the pre-narrex stack and puts it in the current register. BRANCH $I_0, \ldots, I_n$ tests and dispatches on the form of the term represented by the handle in the current register. If the head of this term is a function symbol, the term is pushed on the pre-narrex stack to be eventually narrowed to a head normal form. If it is the special constant `fail`, the current computation is abandoned (see below). If it is a logic variable, control is dispatched to the instruction sub-sequence I_0, which ordinarily arranges to narrow or residuate. Otherwise, the term must be rooted by some constructor c from some datatype t; control is dispatched to instruction sequence I_j, where j is the index of c in the canonical ordering of constructors for t. Note that BRANCH can only occur at the end of an instruction sequence.

$$\mathit{\Gamma}, ([], h{:}N, \_, \_, \_) \Longrightarrow \mathit{\Gamma}, \{(\mathit{code}(f), h{:}N, \_, [], \emptyset)\} \qquad (\mathit{\Gamma}(h) = \langle f, \_\rangle)$$

$$\mathit{\Gamma}, ([], h{:}N, \_, \_, \_) \Longrightarrow \mathit{\Gamma}, \{([], N, \_, [], [])\} \qquad (\mathit{\Gamma}(h) = \langle c, \_\rangle \text{ or } \mathit{\Gamma}(h) = v)$$

$$\mathit{\Gamma}, ([], [], \_, \_, \_) \Longrightarrow \mathit{\Gamma}, \emptyset$$

$$\mathit{\Gamma}, (\textsc{Load}\; p_1 \cdots p_n \;: I, [t_m, \ldots, t_1], \_, T, F) \Longrightarrow \mathit{\Gamma}, \{(I, [t_m, \ldots, t_1], t_m|_{p_1 \cdots p_n}, T, F)\}$$

$$\mathit{\Gamma}, (\textsc{Branch}\; \ldots \;: [], N, h, \_, \_) \Longrightarrow \mathit{\Gamma}, \{([], h{:}N, \_, [], \emptyset)\} \qquad (\mathit{\Gamma}(h) = \langle f, \_\rangle)$$

$$\mathit{\Gamma}, (\textsc{Branch}\; \ldots \;: [], \_, h, \_, \_) \Longrightarrow \mathit{\Gamma}, \emptyset \qquad (\mathit{\Gamma}(h) = \langle \texttt{fail}, ()\rangle)$$

$$\mathit{\Gamma}, (\textsc{Branch}\; I_0, \ldots \;: [], N, h, T, F) \Longrightarrow \mathit{\Gamma}, \{(I_0, N, h, T, F)\} \qquad (\mathit{\Gamma}(h) = v)$$

$$\mathit{\Gamma}, (\textsc{Branch}\; I_0, \ldots, I_n \;: [], N, h, T, F) \Longrightarrow \mathit{\Gamma}, \{(I_j, N, h, T, F)\} \qquad (\mathit{\Gamma}(h) = \langle c, \_\rangle, c\ j\text{-th constructor})$$

$$\mathit{\Gamma}, (\textsc{Push} : I, N, R, T, F) \Longrightarrow \mathit{\Gamma}, \{(I, N, R, R{:}T, F)\}$$

$$\mathit{\Gamma}, (\textsc{Pop} : I, N, \_, t{:}ts, F) \Longrightarrow \mathit{\Gamma}, \{(I, N, t, ts, F)\}$$

$$\mathit{\Gamma}, (\textsc{MakeAnon} : I, N, R, T, F) \Longrightarrow \mathit{\Gamma}[h := v], \{(I, N, R, h{:}T, F)\} \qquad (h \notin \mathit{Dom}(\mathit{\Gamma}), v \text{ fresh})$$

$$\mathit{\Gamma}, (\textsc{StoreVar}\; n : I, N, R, T, F) \Longrightarrow \mathit{\Gamma}[h := v], \{(I, N, R, T, F[n := h])\} \qquad (h \notin \mathit{Dom}(\mathit{\Gamma}), v \text{ fresh})$$

$$\mathit{\Gamma}, (\textsc{MakeVar}\; n : I, N, R, T, F) \Longrightarrow \mathit{\Gamma}, \{(I, N, R, F(n){:}T, F)\}$$

$$\mathit{\Gamma}, (\textsc{MakeTerm}\; s : I, N, R, [t_m, \ldots, t_1], F) \Longrightarrow \mathit{\Gamma}[h := s(t_m, \ldots, t_{m-n+1})], \{(I, N, R, [h, t_{m-n}, \ldots, t_1], F)\} \qquad (h \notin \mathit{Dom}(\mathit{\Gamma}), \mathit{arity}(s) = n \leqslant m)$$

$$\mathit{\Gamma}, (\textsc{Replace} : [], h{:}N, R, [], \_) \Longrightarrow \mathit{\Gamma}[h := R], \{([], h{:}N, \_, [], \emptyset)\}$$

$$\mathit{\Gamma}_0, (\textsc{Narrow} : [], [t_m, \ldots, t_1], h, [c_1, \ldots, c_k], \_) \Longrightarrow \mathit{\Gamma}_k, \{([], [h_i], \_, [], \emptyset) \mid 1 \le i \le k\}$$
$$\text{where } \sigma_i = [h := c_i] \text{ and } (\mathit{\Gamma}_i, h_i) = \mathit{clone}_{\sigma_i}(\mathit{\Gamma}_{i-1}, t_1) \quad (1 \le i \le k)$$

$$\mathit{\Gamma}_0, (\textsc{Choice} : [], [t_m, \ldots, t_1], \_, [c_1, \ldots, c_k], \_) \Longrightarrow \mathit{\Gamma}_k, \{([], [h_i], \_, [], \emptyset) \mid 1 \le i \le k\}$$
$$\text{where } \sigma_i = [t_m := c_i] \text{ and } (\mathit{\Gamma}_i, h_i) = \mathit{clone}_{\sigma_i}(\mathit{\Gamma}_{i-1}, t_1) \quad (1 \le i \le k)$$

Fig. 1. Machine instruction set. Instructions map a heap and an active computation to a revised heap and a set of result computations. Computations are described by tuples of the form (I, N, R, T, F), where I is an instruction sequence, N is the pre-narrex stack, R is the current register, T is the pre-term stack, and F is the free variable map. $\mathit{code}(f)$ denotes the sequence of virtual machine instructions associated to function f as described in Section 3.7. Standard Haskell-style list notation is used for stacks and sequences. An underscore ($\_$) denotes a field whose contents don't matter.

A number of instructions manipulate the pre-term stack. PUSH and POP move handles between the current register and the stack. MAKEANON creates a fresh, independent free variable in the heap and pushes its handle. MAKEVAR pushes the handle of a (potentially) shared free variable (previously created by STOREVAR) from the shared free-variable map. MAKETERM s constructs a new term representation in the heap with root symbol s and the top $arity(s)$ elements of the stack as arguments, and pushes its handle in place of the arguments. Finally, REPLACE updates the handle on the top of *pre-narrex* stack to have the same contents as the handle in the current register.

The remaining instructions, which only appear at the end of an instruction sequence, place multiple, non-deterministic alternative computations into the active pool. NARROW executes a narrowing step. When this instruction is executed, the current register holds the handle for a variable v and and the pre-term stack holds handles for the instantiations $c_1, \ldots, c_k$, $k > 0$, of this variable. For each instantiation c_i, the root term of the computation t_1 is cloned under the substitution $[h := c_i]$. The computation executing the non-deterministic step is abandoned and a new computation corresponding to each clone is added to the pool. Note that each new computation starts from the root term and an empty pre-narrex stack; this stack gets rebuilt independently in each computation. CHOICE is similar, except that it executes a non-deterministic reduction step. When this instruction is executed, the top of the pre-narrex stack holds a narrex t_m and the pre-term stack holds the replacements $c_1, \ldots, c_k$, $k > 1$, of this narrex. For each replacement c_i, the root term of the computation t_1 is cloned under the substitution $[t_m := c_i]$.

There is one further instruction, RESIDUATE, which moves a computation from the active pool to a waiting pool pending the instantiation of a logic variable. A precise description of this instruction and of the remainder of the residuation mechanism are beyond the scope of this paper.

In addition to these instructions, some activities of the machine are performed by built-in functions. Generally, these are library functions that could not be defined by ordinary rewrite rules. An example of a built-in function is *apply*, which takes two terms as arguments and applies the first to the second. For correctly-typed programs, the first argument of *apply* evaluates to a term of the form $f(x_1, \ldots, x_n)$ where the arity of f is greater than n, i.e., f is a partial application. The function *apply* performs a simple manipulation of the representation of terms. It would be easy to replace the built-in function *apply* with a machine instruction. However, built-in functions are preferable to machine instructions because they keep the machine simpler and they are loaded only when needed.

Figure 2 shows the code for the list concatenation function "++" defined in (4). This code is executed when the top of the pre-narrex stack contains a term of the form u++v.

3.7 Compilation

Every function of an overlapping inductively sequential program has a *definitional tree* [2, 3], which is a hierarchical representation of the rewrite rules of a function that has become the standard device for the implementation of narrowing computations. We compile each definitional tree into a sequence of virtual machine instructions. Because

```
 1  LOAD         1     load u in the current register
 2  BRANCH
      [                u is an uninstantiated variable
 3      MAKETERM  []    pre-term stack contains [ ]
 4      MAKEANON        push _
 5      MAKEANON        push _
 6      MAKETERM  :     pre-term stack contains [ ] and _ : _
 7      NARROW
      ]
      [                u is [ ]
 8      LOAD      2     load v
 9      REPLACE
      ]
      [                u is u0 : us
10      LOAD      2     load v
11      PUSH
12      LOAD      1·2   load us
13      PUSH
14      MAKETERM  ++    pre-term stack contains us++v
15      LOAD      1·1   load u0
16      PUSH
17      MAKETERM  :     pre-term stack contains u0 : us++v
18      POP
19      REPLACE
      ]
```

Fig. 2. Compilation of the definition of the function "++". This code is executed to evaluate a term of the form u++v. The instruction numbers at the left and the comments at the right are not part of the code itself.

a definitional tree is a high-level abstraction for the definition of a sound, complete and theoretically efficient narrowing strategy [6], mapping this strategy into virtual machine instructions increases our confidence in both the correctness and the efficiency of the execution. The notation for the variant of definitional trees we use is summarized in Figure 3.

A trees consist of internal *Branch* nodes, which encode choices between left-hand-side patterns of rewrite rules, and leaf *Rule* nodes, which correspond to the right-hand sides of rewrite rules. *Branch* nodes contain a pattern p to match, a position *pos* within the term to be matched, a flag *flex?* indicating whether or not the branch is *flexible* or *rigid*, i.e., whether to narrow or residuate if the corresponding position of a term being

$$
\begin{array}{rcl}
\text{(definitional tree) } \mathcal{T} & = & \textit{Branch}(p, \textit{pos}, \textit{flex?}, [\mathcal{T}_1, \ldots, \mathcal{T}_n]) \\
 & | & \textit{Rule}(p, [r_1, \ldots, r_n]) \\
 & & \\
\text{(right-hand side) } r & = & ([v_1, \ldots, v_n], t)
\end{array}
$$

Fig. 3. Notation for definitional trees.

processed is a variable. In the node *Rule(p,rs)*, *rs* is a list of non-deterministic alternative right-hand sides for the rule. Each right-hand side (vs, t) consists of a term t and a list of free variables *vs* that appear in t but not in p.

As examples, the definitional tree for the function `(++)` defined in (4) is:

$$\begin{aligned}&\textit{Branch}(\texttt{x++y},\texttt{1},\textit{True}, [\textit{Rule}(\texttt{[]++y}, [([], \texttt{y})]),\\ &\qquad\qquad\qquad\qquad \textit{Rule}(\texttt{(x:xs)++y}, [([], \texttt{x:(xs++y)})])],\end{aligned}$$

the tree for `palind` (5) is:

$$\textit{Rule}(\texttt{palind s}, [([\texttt{x}, \texttt{y}], \texttt{s =:= x++(y:reverse x)})]),$$

and the tree for `digit` (1) is:

$$\textit{Rule}(\texttt{digit}, [([], \texttt{0}), ([], \texttt{1}), \ldots, ([], \texttt{9})]),$$

where, for readability, we write terms and patterns using infix notation.

Figure 4 gives an algorithm for compiling definitional trees to sequences of abstract machine instructions. For simplicity, we assume all definitional trees are canonical, in the sense that every *Branch* node corresponding to a position of type τ has a child for each data constructor of τ, and the children are in the canonical order for data constructors. (In reality, the compiler would use auxiliary type information to determine the full set of possible children, and generate code to produce `fail` for the missing ones.) We assume the existence of a function *posOf* v p that returns the position (if any) of variable v in pattern p (assuming v appears at most once in p). Various optimizations on the resulting code are possible; for example, the sequence of instructions [PUSH,POP] can be omitted, as illustrated by the code in Figure 2, or the instructions STOREVAR n and MAKEVAR n can be replaced by a single MAKEANON instruction for free variables that occur only once in the right-hand side.

Some practical adjustments to the pseudo-code of Figure 4 are necessary to accommodate built-in types, such as integers and characters. There are a few additional machine instructions, e.g., MAKEINT and MAKECHAR, for this purpose.

4 Implementation

We have two prototype implementations of the virtual machine described in this paper. One implementation, in Java, is currently our main development avenue. A second implementation, in Standard ML, is being used mostly as a proof of concept. Since the code is not optimized because it is still evolving, we do not present a detailed benchmark suite here. Nevertheless, the initial performance results appear to be promising. A computationally intensive test computes Fibonacci numbers with an intentionally inefficient program. This test shows that the machine executes approximately 0.5 million reductions (i.e., function calls) per second on a 2.0 Ghz Linux-PC (with AMD Athlon XP 2600). On the same benchmark, the PAKCS [14] implementation of Curry, which compiles Curry programs into Prolog using the scheme in [7], runs about twice as fast. PAKCS is one of the most efficient Curry implementations, apart from MCC [20], which produces native code. However, neither of these implementations is operationally complete. For example, neither produces a solution to example (3).

```
$compileTree\ (Branch(p, pos, flex?, [\mathcal{T}_1, \ldots, \mathcal{T}_n]))$ =
  [LOAD $pos$,
   BRANCH [$handleVariable$,
           $compileTree\ \mathcal{T}_1$,
           ...,
           $compileTree\ \mathcal{T}_n$]]
   where $handleVariable$ =
        if $flex?$ then
            $buildChoice_1$ ++ $\cdots$ ++ $buildChoice_n$ ++ [NARROW]
                where $buildChoice_i$ = [MAKEANON$_1$,...,MAKEANON$_{n_i}$,
                                         MAKETERM $c_i$]
                    where $c_i(d_1, \ldots, d_{n_i})$ = $(patternOf\ \mathcal{T}_i)|_{pos}$
        else [RESIDUATE]

$compileTree\ (Rule(p, [rhs_1, \ldots, rhs_n])$ =
  if $n = 1$ then
      ($compileRhs\ rhs_1$) ++ [POP,REPLACE]
  else ($compileRhs\ rhs_1$) ++ ... ++ ($compileRhs\ rhs_n$) ++ [CHOICE]
  where $compileRhs\ ([v_1, \ldots, v_n], t)$ =
                [STOREVAR 1,...,STOREVAR $n$] ++ ($compileTerm\ t$)
      where $compileTerm\ (v)$ = if $\exists j\ with\ v = v_j$ then
                                     [MAKEVAR $j$]
                                 else [LOAD ($posOf\ v\ p$),PUSH]
            $compileTerm\ (s(t_1, \ldots, t_n))$ =
              ($compileTerm\ t_n$) ++ $\cdots$ ++ ($compileTerm\ t_1$) ++
              [MAKETERM $s$]
```

Fig. 4. Pseudo-code for compilation of definitional trees to sequences of virtual machine instructions. Standard Haskell-style notation is used for lists.

We have used Java and ML due to their built-in support for automatic memory management and appropriate programming abstractions which simplified the development of our prototypes. The same approach has been taken in [16], which describes an abstract machine for Curry and its implementation in Java. On the negative side, the use of Java limits the speed of the execution—the Java implementation in [16] is more than an order of magnitude slower than PAKCS [7]. On the positive side, our machine can be also implemented in C/C++ from which we can expect a considerable efficiency improvement.[3] A possible strategy is to integrate a C-based execution engine into the Java support framework.

Non-deterministic computations are executed independently. However, because of the use of term handles, a common deterministic term of two independent computations is evaluated only once. For example, consider the term `digit` $+ t$, where `digit` is defined in (1). A distinct computation is executed for each replacement of `digit`, but t

[3] [16] compares the speed of the same virtual machine for Curry coded in Java vs. in C/C++. The latter is more than one order of magnitude faster compared to a Java implementation with a Just-In-Time compiler.

is evaluated only once for all these computations. In situations of this kind, our machine is faster than PAKCS.

In our implementations, a narrex is replaced in place (with a destructive update) whenever possible. Non-deterministic steps prevent replacement in place, since several replacements should update a single term. Currently, the machine constructs not only the replacement of a narrex, but also the spine of the entire term in which the narrex occurs. This is unnecessarily inefficient and we plan to improve the situation in the future together with other optimizations of the machine architecture and code.

Our virtual machine is intended for the execution of functional logic programs in a variety of source languages. Our immediate choice of source language is Curry [18]. For this application, we have a complete compiler (written in Curry) into our virtual machine but several other non-trivial software components, such as a command line parser, a loader, a debugger and a run-time library, are necessary as well. The virtual machine has good built-in capabilities for tracing and debugging. A specific problem of an operationally complete implementation of non-deterministic computations is that steps of different computations are interleaved. Presenting steps in the order in which they are executed produces traces which are hard to read. An external debugger with a suitable interface for non-deterministic computations is described in [9]. Finally, we have implemented a handful of modules for built-in types, such as the integers, that cannot be compiled from source programs.

To conclude, we have a solid, though preliminary, implementation of the virtual machine. Several key software components of an interactive development environment need further work. The Java implementation of the machine is available for download from `http://redstar.cs.pdx.edu/~antoy/flp/vm`. The distribution also links a tutorial description of the machine including an animation of the behavior of the instructions.

5 Conclusion and Related Work

We have described the architecture of a virtual machine for the execution of functional logic computations. The machine's design is based on solid theoretical results. In particular, the machine is intended for overlapping inductively sequential programs and computes only root-needed steps (modulo non-deterministic choices). Larger classes of programs, up to those modeled by the whole class of constructor-based conditional rewrite systems, can be executed after initial transformation.

A small set of machine instructions performs pattern matching and narrex replacement, two key activities of the machine. Both narrowing and non-deterministic steps are executed by a single instruction since the machine is specifically designed for functional logic computations. The machine is also designed to execute several computations concurrently to ensure the operational completeness. Implementations of the machine in Java and ML are complete and fairly efficient, through not yet optimized.

The implementation of functional logic languages is an active area of research. A common approach is the translation of functional logic source programs into Prolog programs, where Prolog has the role of a portable, specialized machine language, e.g., [7]. Another approach relies on an abstract machine. The machine presented here is only one of several alternatives the authors have considered. In [8], Antoy, Hanus, et al. describe a virtual machine with many similarities to that described in this paper, but a major difference. Functions are compiled into Java objects rather than sequences of virtual machine instruction as in the example of Figure 2, i.e., the target language is Java rather than an instruction set of a virtual machine. In [16] Hanus and Sadre presented also a virtual machine for compiling Curry programs that exploits Java threads to implement the concurrent features of Curry and ensures the operational completeness of non-deterministic computations. To manage the bindings of logical variables caused by different non-deterministic computations, they used bindings tables that are partially shared between computations. The resulting architecture is more complex than the machine presented in this paper and has fewer possibilities for optimization, e.g., the sharing of deterministic evaluations between non-deterministic computations discussed in Section 4. Thus, this implementation is no longer supported.

In [22], Tolmach, Antoy, and Nita describe a definitional interpreter for Curry-like languages based on the semantics of Albert, *et al.* [1]. The primary contrast with the present work is in the treatment of the heap. Rather than conceiving of the system as a graph rewriting engine that generates modified copies of the source term as it runs, [22] treats the program as fixed, read-only code that operates on multiple variant *versions* of the heap. A direct performance comparison between these two approaches remains to be made.

Among related work by others, Chakravarty and Lock [10] proposed a virtual machine for functional logic languages that combines implementation techniques from functional and logic programming in an orthogonal way. To implement logic language features, they used traditional logic programming implementation techniques based on backtracking so that the operational completeness is not ensured. The same is true for the virtual machine used in the Curry implementation MCC [20]. Due to the native code compilation used in MCC, the implementation is quite efficient but not operational complete due to the use of a backtracking strategy.

A minimal comparison of efficiency was addressed earlier. However, our effort is mainly characterized by the simplicity of both the instruction set and the storage areas and by the rigorous theoretical results on which the machine is founded.

Acknowledgments

Pravin Damle made extensive contributions to the implementation. Marius Nita and the anonymous reviewers gave helpful suggestions on the presentation of the paper.

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation (to appear)*, 2005.
2. S. Antoy. Definitional trees. In *Proc. 3rd International Conference on Algebraic and Logic Programming (ALP'92)*, pages 143–157. Springer LNCS 632, 1992.
3. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. Int. Conf. on Algebraic and Logic Programming (ALP'97)*, pages 16–30. Springer LNCS 1298, 1997.

4. S. Antoy. Constructor-based conditional narrowing. In *Proc. of the 3rd International Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 199–206, Florence, Italy, Sept. 2001. ACM.
5. S. Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 2005. To appear.
6. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.
7. S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into Prolog. In *Proc. of the 3rd International Workshop on Frontiers of Combining Systems (FroCoS 2000)*, pages 171–185, Nancy, France, March 2000. Springer LNCS 1794.
8. S. Antoy, M. Hanus, B. Massey, and F. Steiner. An implementation of narrowing strategies. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, pages 207–217. ACM Press, 2001.
9. S. Antoy and S. Johnson. TeaBag: A functional logic language debugger. In *Proc. 13th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2004)*, pages 4–18, Aachen (Germany), 2004. Technical Report AIB-2004-05, RWTH Aachen.
10. M.M.T. Chakravarty and H.C.R. Lock. Towards the uniform implementation of declarative languages. *Computer Languages*, 23(2-4):121–160, 1997.
11. J. C. González Moreno, F. J. López Fraguas, M. T. Hortalá González, and M. Rodríguez Artalejo. An approach to declarative programming based on a rewriting logic. *The Journal of Logic Programming*, 40:47–87, 1999.
12. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
13. M. Hanus. A unified computation model for functional and logic programming. In *Proc. 24st ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 80–93, 1997.
14. M. Hanus, S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2004.
15. M. Hanus and C. Prehofer. Higher-order narrowing with definitional trees. *Journal of Functional Programming*, 9(1):33–75, 1999.
16. M. Hanus and R. Sadre. An abstract machine for Curry and its concurrent implementation in Java. *Journal of Functional and Logic Programming*, 1999(6), 1999.
17. M. Hanus and F. Steiner. Controlling search in declarative programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, pages 374–390. Springer LNCS 1490, 1998.
18. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8). Available at `http://www.informatik.uni-kiel.de/~curry`, 2003.
19. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
20. W. Lux and H. Kuchen. An efficient abstract machine for Curry. In K. Beiersdörfer, G. Engels, and W. Schäfer, editors, *Informatik '99 — Annual meeting of the German Computer Science Society (GI)*, pages 390–399. Springer Verlag, 1999.
21. A. Middeldorp. Call by need computations to root-stable form. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 94–105, Paris, 1997.
22. A. Tolmach, S. Antoy, and M. Nita. Implementing functional logic languages using multiple threads and stores. In *Proc. of the Ninth International Conference on Functional Programming (ICFP 2004)*, pages 90–102, Snowbird, Utah, USA, Sept. 2004. ACM Press.
23. D.H.D. Warren. Higher-order extensions to PROLOG: are they needed? In *Machine Intelligence 10*, pages 441–454, 1982.