Architecture of 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.

[2] Christian-Albrechts-Universität Kiel, Institut für Informatik
Olshausenstr. 40, D-24098 Kiel, Germany.

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, represented as definitional trees, to sequences of machine instructions. The machine has been implemented in Java and in Standard ML.

1 Introduction

Functional logic programming aims at integrating into a single paradigm the characteristic features of functional and logic programming. In the last decade, the theory of functional logic computations has made substantial progress. Significant milestones include a model that integrates narrowing and residuation [12], 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, 10].

These results have been influential in the design and implementations of functional logic programming languages, e.g., Curry [16] and $\mathcal{TOY}$ [17]. Most existing implementations of these languages are based on a translation of source code to Prolog code (e.g. as described in [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 encapsulated search strategies [15].

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 describes 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

* 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 grant Ha 2457/1-2.

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, represented as a definitional tree, 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 [11] 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 program 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 distinct values. For example, a program that solves a *cryptarithm* must compute a mapping from letters to digits. This can be expressed as:

```
mapping letter = digit

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 portion of the program. Non-determinism broadens the class of programs that can be coded using functional composition [3]. 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 list | list =:= x++[e] = e  where x,e free                    (2)
```

The evaluation of `(last [1,2,3])` attempts to verify that the instantiated condition `[1,2,3] =:= x++[e]` holds. 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. Although in this example the computation is deterministic, i.e., (`last` *list*) has at most one value for any list *list*, typical narrowing computations are non-deterministic since they involve some guessing.

Residuation, similar in intent to narrowing, handles the evaluation of an expression containing an uninstantiated variable. In this case, though, the evaluation of the expression suspends. 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 made at compile time 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. Also, this class properly includes the first-order programs of the functional languages ML and Haskell. Also, 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 [20]. (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 the features of functional logic computations shape the architecture of our virtual machine.

3.1 Pool of Computations

A fundamental aspect of functional logic computations is (don't know) 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 strict narrowing or rewriting.) For example, in the cryptarithm solver mentioned earlier, the evaluation of `mapping 'S'`, assuming that `'S'` is a letter of the cryptarithm, leads to 10 possible replacements.

One of our main goals is to ensure the operational completeness of computations. The simplest policy to ensure this 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*, *residuating*, etc.,

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 choice, 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 execute 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 internal 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 one or two 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 allocate resources to active computations so that they make some "progress" toward a result over time. We considered several strategies to ensure a fair sharing of resources. 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. An analogous strategy could allocate a fixed number of virtual machine instructions, instead.

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

$$
\begin{array}{rl}
\text{(variable)} & v \\
\text{(constructor)} & c \\
\text{(function)} & f \\
\text{(symbol)} & s = c \mid f \\
\text{(term)} & t = v \mid s(t_1, \ldots, t_n) \\
\text{(data term)} & d = v \mid c(d_1, \ldots, d_n) \\
\text{(pattern)} & p = f(d_1, \ldots, d_n)
\end{array}
$$

Fig. 1. Notation for terms and patterns.

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 [12], 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 is a variable or a symbol of fixed arity $n \geqslant 0$ applied to n terms; symbols are partitioned into data constructors and functions. Figure 1 summarizes our notation for terms. In examples, we often write terms using infix notation for symbols. A position *pos* in a term is represented by a sequence of natural numbers representing subterm choices, beginning at the root. For example, the position of `x` in `f(y,b(x,z))` is the sequence `[1,0]`. 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, i.e., a term without defined functions. The computed answer is a substitution, possibly the identity, from some free variables of the term being evaluated to data terms. For example, the evaluation of `[1,2,3] =:= x++[e]`, in example (2), returns the computed value `Success`, a predefined constant for constraints, and the computed answer $\{\texttt{x} \mapsto \texttt{[1,2]},\ \texttt{e} \mapsto \texttt{3}\}$.

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 find 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 hosting 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, 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. Executing these computations is the core activity of the machine, during which the definitions of functions are applied. A single normal form computation typically spawns multiple

head normal form computations over its sub-terms, as described in Section 3.3. Head normal form computations are described in more detail in Section 3.4.

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 [12]). 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*.

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

3.3 Needed Sequential Steps

We chose overlapping inductively sequential programs as the source language for our machine because they have useful properties that allow efficient implementation. Loosely speaking, in any term, it is easy either to find a step that "must" be executed or to determine that the computation is complete—either successfully or unsuccessfully. The step, possibly modulo a non-deterministic choice, cannot be avoided to obtain a result of the computation, a desirable property referred to as *need*, and it is computed by looking only at the term, a desirable property referred to as *sequentiality*.

This need-based strategy applies to terms whose leading symbol is a function. Within the virtual machine, it is implemented by the execution mechanism for head normal form computations (see next section). If the leading symbols of a term is a data constructor, this top portion of the term will not change during the rest of the computation and, in principle, it could be output. In this case the strategy is applied to each *maximal* subterm led by a function. If several such subterms exist, all must be evaluated and the results are independent of the evaluation order. Responsibility for managing these sub-evaluations belongs to the execution mechanism for normal form computations.

3.4 Head Normal Form Computations

The execution of a head normal form computation attempts to rewrite a *function*-rooted term into a *constructor*-rooted term. The evaluation strategy executed by our machine is *root-needed* reduction [19] with the addition of narrowing and non-deterministic steps. Simply put, the strategy attempts to repeatedly apply rewrite rules at the top of a function-rooted term until a constructor-rooted term is obtained.

This strategy can be applied independently for each function defined in the source program. The implementation of the 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 a function-rooted term. This can occur for one of only two reasons: (1) a function-rooted argument of a function application must be evaluated to a constructor-rooted term 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                                   (3)
(x:xs) ++ y = x : xs ++ y
```

The term $t =$ `head (`u `++` v`)`, for any u and v, is an example of the first condition. To evaluate t, it is necessary to evaluate `(`u `++` v`)` which is a recursive instance of the original problem, i.e., to evaluate a function-rooted term to a constructor-rooted term. The virtual machine deals with possibility within the implementation of the BRANCH instruction, which tests and dispatches on the form of a term.

The term $t =$ `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.

3.5 Data Representation and Storage Areas

Each term manipulated by the virtual machine is represented as an immutable heap-allocated record. These records are referenced indirectly via *term handles*, which contain mutable pointers to terms. The representation record for an application term contains the applied symbol together with pointers to the handles of the argument subterms. During computation, the machine holds a pointer to the handle of the current narrex (as described in more detail below). When the narrowed result term has been computed, the contents of the narrex handle are overwritten with a pointer to that result. This has the effect of updating the original narrex wherever it appears as a subterm anywhere in the heap.

Substitutions are stored as finite maps from variables names to term handles. They are applied to terms by creating a clone (deep copy) of the term in which each variable in the domain of the substitution is replaced by the corresponding image term. Substitutions are never applied destructively to change a term in-place.

As discussed in the previous sections, our machine fairly executes a pool of independent computations. The context of each computation includes four separate *storage areas*, a generic name for stacks and registers. Stacks and registers hold pointers to handles for terms; more loosely, we just say they hold terms.

Suppose that t is the term to evaluate in a head-normal form computation. We recall that initially t is function-rooted; the computation completes successfully when t is evaluated to a constructor-rooted term. 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 function-rooted subterms of t. The **pre-narrex stack** keeps track of these recursive computations. It is a stack containing (pointers to handles for) terms $t_1, t_2, \ldots, t_n$, with t_1 the bottom, having the following properties.

1. At the beginning of the computation, $n = 1$ and $t_1 = t$.
2. Every term, with the possible exception of t_n, the top of the stack, is function-rooted and it is not a narrex.
3. For all i, t_{i+1} is a subterm of t_i with the property that t_{i+1} must be evaluated to a constructor-rooted term before t_i can be evaluated.

The top of the pre-narrex stack contains the term currently being evaluated. Referring to example (3), if `head (`u `++` v`)` is on the pre-narrex stack, then u `++` v will be pushed on the stack, too, because the former cannot be evaluated to a constructor-rooted term unless the latter is evaluated to a constructor-rooted term. 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. Many of the machine's instructions implicitly reference this register. For example, to apply a rewrite rule of the function "`++`" defined in (3) to the term u `++` 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 terms 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 terms `[3,4]` and `[2]` are pushed on the pre-term stack. Executing MAKETERM `++` replaces them with `[2]++[3,4]`. Then, the term `1` is pushed on the stack as well and executing MAKETERM `:` replaces the two topmost elements with `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, if a free variable has multiple occurrences in an term, when the variable is created, a reference to it is stored in a register to be later retrieved for each occurrence. 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          (4)
```

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 again on the stack. Thus, a reference to `x` must be kept around so that it can be retrieved later and pushed again. The machine has an (arbitrarily large) set of registers exclusively for storing free variables that must be used several times.

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. There are a dozen or so different instructions. Some instructions move information between the various storage areas. Others build or take apart terms.

A computation starts with a single term in the pre-narrex stack. The information in all the other storage areas is irrelevant. The machine repeats the following cycle. If the pre-narrex stack is empty, the computation is completed. If the top of the pre-narrex stack is a constructor-rooted term, the stack is popped. If the top of the pre-narrex stack is a function-rooted term, and f is the root symbol, the machine retrieves the code of f and executes it.

Some instructions are parameterized by (references to) symbols. In the examples, we represent these symbols with their symbolic names, but internally symbols are represented by integers that index an array. Each entry of the array contains run-time information about a symbol, such as its printable name and (for functions) its code.

Figure 2 gives transition rules for the instructions that act only on the storage areas. The remaining important instructions, which only appear at the end of a code sequence, act on the computation pool:

Replace: Update the handle on the top of the pre-narrex stack to point to the term in the current register; put this computation back into the computation pool.

Narrow: Execute a narrowing step. When this instruction is executed, the current register holds a variable and the pre-term stack holds one or more instantiations of this variable. Each instantiation implicitly defines a substitution, which is applied to the base term. A new normal form computation is added to the pool for each resulting version of the base term. The current computation is abandoned.

Choice: Execute non-deterministic step. When this instruction is executed, the current register holds a narrex and the pre-term stack holds one or more replacements of this narrex. For each replacement, a new version of the base term is created, and a corresponding new normal form computation is added to the pool. The current computation is abandoned.

$$
\begin{array}{rll}
(\textsc{Load}\ p_0,\ldots,p_n\ : I,[t_1,\ldots,t_m],R,T,F) & \Longrightarrow (I,[t_1,\ldots,t_m],t_m|_{p_0,\ldots,p_n},T,F) & \\
(\textsc{Branch}\ I_0,\ldots,I_n\ : [],N,R,T,F) & \Longrightarrow ([],N\texttt{++}[R],R,T,F) & \text{if } R=f(\cdots)\\
(\textsc{Branch}\ I_0,\ldots,I_n\ : [],N,\texttt{fail},T,F) & \Longrightarrow \textit{abandon computation} & \\
\textsc{Branch}\ I_0,\ldots,I_n\ : [],N,v,T,F) & \Longrightarrow (I_0,N,R,T,F) & \\
(\textsc{Branch}\ I_0,\ldots,I_n\ : [],N,c(\cdots),T,F) & \Longrightarrow (I_j,N,c(\cdots),T,F) & \text{if } c \text{ has index } j\\
(\textsc{Push} : I,N,R,T,F) & \Longrightarrow (I,N,R,T\texttt{++}[R],F) & \\
(\textsc{Pop} : I,N,R,[t_1,\ldots,t_m],F) & \Longrightarrow (I,N,t_m,[t_1,\ldots,t_{m-1}],F) & \\
(\textsc{MakeAnon} : I,N,R,T,F) & \Longrightarrow (I,N,R,T\texttt{++}[v],F) & \text{where } v \text{ fresh}\\
(\textsc{StoreVar}\ n : I,N,R,T,F) & \Longrightarrow (I,N,R,T,F[n\leftarrow v]) & \text{where } v \text{ fresh}\\
(\textsc{MakeVar}\ n : I,N,R,T,F) & \Longrightarrow (I,N,R,T\texttt{++}[F(n)],F) & \\
(\textsc{MakeTerm}\ s : I,N,R,[t_1,\ldots,t_m],F) & \Longrightarrow & \\
& \quad (I,N,R,[t_1,\ldots,t_{m-n},s(t_m\ldots,t_{m-n+1})],F) & \text{where } \mathit{arity}(s)=n
\end{array}
$$

Fig. 2. Machine instruction set. Tuples are of the form (I, N, R, T, F), where I is an instruction sequence, N is the pre-narrex stack, T is the pre-term stack, R is the current register, and F is the free variable registers. Standard Haskell-style notation is used for lists.

RESIDUATE: The computation state is changed to *residuating*, and it is returned to the pool.

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 3 shows the code of the polymorphic function "`++`" which concatenates two lists. This code is executed when the top of the pre-narrex stack contains a term of the form u`++`v. Note that some sequences of instructions tend to occur frequently in compiled code. It would be easy to improve the performance of the machine by introducing new combined instructions to replace such sequences, but we do not describe this here.

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 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.

```
 1  LOAD            0       load u in the current register
 2  BRANCH
      [                     u is an uninstantiated variable
 3     MAKETERM     []      pre-term stack contains []
 4     MAKEANON             push _
 5     MAKEANON
 6     MAKETERM     :       pre-term stack contains [] and _:_
 7     NARROW
      ]
      [                     u is []
 8     LOAD         1       load v
 9     REPLACE
      ]
      [                     u is u0:us
10     LOAD         1       load v
11     PUSH
12     LOAD         0,1     load us
13     PUSH
14     MAKETERM     ++      pre-term stack contains us++v
15     LOAD         0,0     load u0
16     PUSH
17     MAKETERM     :       pre-term stack contains u0:us++v
18     POP
19     REPLACE
      ]
```

Fig. 3. 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.

The notation for the variant of definitional trees we use is summarized in Figure 4. *Branch* nodes contain a flag 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 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.

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

Fig. 4. Notation for definitional trees.

As examples, the definitional tree for the function `(++)` defined in (3) is:

Branch `(x++y,[0],` *True,*
[*Rule*(`[]++y`, [([], `y`)]),
Rule(`(x:xs)++y`, [([], `x:(xs++y)`)])],

the tree for `palind` (4) is:

Rule(`palind s`, [([`x`, `y`], `s =:= x++(y:reverse x)`)]),

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

Rule(`digit`, [([], `0`), ([], `1`), ..., ([], `9`)]),

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

Figure 5 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 of data constructor of τ, and the children are in the canonical order for data constructors. (In reality, the compiler would tolerate missing children and generate code to produce `fail` for them; it would use auxiliary type information to determine the full set of possible children.) 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 is illustrated by the code in Figure 3, or the instructions STOREVAR n and MAKEVAR n can be replaced by a single MAKEANON instruction for free variables with only one occurrence in the right-hand side.

Some practical adjustments to the pseudo-code of Figure 5 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 [13] 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 [18], which produces native code. However, note that our implementation is operationally complete, i.e., there are programs where our implementation computes a result in contrast to [7].

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 [14], which describes an abstract machine

```
compileTree (Branch(p, pos, flex?, [T_1, ..., T_n])) =
  [LOAD pos,
   BRANCH [handleVariable,
           compileTree T_1,
           ...,
           compileTree T_n]]
   where handleVariable =
           if flex? then
               buildChoice_1 ++ ··· ++ buildChoice_n ++ [NARROW]
                   where buildChoice_i = [MAKEANON_1, ..., MAKEANON_{n_i},
                                          MAKETERM c_i]
                         where c_i(d_1, ..., d_{n_i}) = (patternOf T_i) |pos
           else [RESIDUATE]

compileTree (Rule(p, [rhs_1, ..., rhs_n]) =
  if n == 1 then
      (compileRhs rhs_1) ++ [POP,REPLACE]
  else (compileRhs rhs_1) ++ ... ++ (compileRhs rhs_n) ++ [CHOICE]
  where compileRhs ([v_0, ..., v_n], t) =
                  [STOREVAR 0, ..., STOREVAR n] ++ (compileTerm t)
        where compileTerm (v) = if ∃j s.t. v = v_j then
                                    [MAKEVAR j]
                                else [LOAD (posOf v p), PUSH]
              compileTerm (s(t_1, ..., t_n)) =
                (compileTerm t_n) ++ ··· ++ (compileTerm t_1) ++
                [MAKETERM s]
```

Fig. 5. Pseudo-code for compilation of definitional trees to sequences of virtual machine instructions. Standard Haskell-style notation is used for lists and list concatenation (`++`).

for Curry and its implementation in Java. On the negative side, the use of Java limits the speed of the execution—the Java implementation [14] 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 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

[3] [14] 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.

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 [16]. 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. At the time of writing (Sept. 2003), our Java implementation provides both a loader and a command line interpreter. The virtual machine has good built-in capabilities for tracing and debugging, but a suitable user interface is needed. 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. A suitable graphical user interface would make these traces easier to read. Finally, we have implemented only 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 prototypical, 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 instructions 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 source Prolog programs, where Prolog has the role of a portable, specialized machine language, e.g., [7]. Another approach relies on an abstract machine. A detailed comparison of several recent related attempts [8, 9, 14, 18] would go well beyond the space constraints of this paper. A minimal comparison of efficiency was addressed earlier. Our effort is 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.

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for functional logic languages. *Electronic Notes in Theoretical Computer Science*, 76, 2002.
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. In Bernhard Gramlich and Salvador Lucas, editors, *Electronic Notes in Theoretical Computer Science*, volume 57. Elsevier Science Publishers, 2001.
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. M.M.T. Chakravarty and H.C.R. Lock. Towards the uniform implementation of declarative languages. *Computer Languages*, 23(2-4):121–160, 1997.
10. 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.
11. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
12. 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.
13. M. Hanus, S. Antoy, 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/`, 2000.
14. 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.
15. 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.
16. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8). Available at `http://www.informatik.uni-kiel.de/~curry`, 2003.
17. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
18. 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.
19. 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.
20. D.H.D. Warren. Higher-order extensions to PROLOG: are they needed? In *Machine Intelligence 10*, pages 441–454, 1982.