An Abstract Machine for Curry and its Concurrent Implementation in Java[*]

Michael Hanus[†] Ramin Sadre[‡]

March 30, 1999

Abstract

Curry is a multi-paradigm declarative language aiming to amalgamate functional, logic, and concurrent programming paradigms. Curry combines in a seamless way features from functional programming and (concurrent) logic programming. Curry's operational semantics is based on the combination of lazy reduction of expressions together with a possibly non-deterministic binding of free variables occurring in expressions. Moreover, (equational) constraints can be executed concurrently which provides for passive constraints and concurrent computation threads that are synchronized on logical variables. This paper describes in an object-oriented style an abstract machine for executing Curry programs. The machine is designed to provide a link for compiling Curry programs into Java but it can also be a basis for implementations of Curry in other (object-oriented) languages. The main emphasis of the Java-based implementation is the exploitation of Java threads to implement the concurrent and non-deterministic features of Curry.

Keywords: Functional logic programming, lazy evaluation, concurrency, implementation

[*]This research has been partially supported by the German Research Council (DFG) under grant Ha 2457/1-1.

[†]Informatik II, RWTH Aachen, D-52056 Aachen, Germany, hanus@informatik.rwth-aachen.de.

[‡]Department of Computer Science—Distributed Systems, RWTH Aachen, D-52056 Aachen, Germany, ramin@lvs.informatik.rwth-aachen.de.

1 Introduction

Curry [13, 16] is a multi-paradigm declarative language aiming to integrate functional, logic, and concurrent programming paradigms. Curry combines in a seamless way features from functional programming (nested expressions, lazy evaluation, higher-order functions), logic programming (logical variables, partial data structures, built-in search), and concurrent programming (concurrent evaluation of expressions with synchronization on logical variables). Moreover, Curry provides additional features in comparison to the pure languages (compared to functional programming: search, computing with partial information; compared to logic programming: more efficient evaluation due to the deterministic and demand-driven evaluation of functions). Moreover, it also amalgamates the most important operational principles developed in the area of integrated functional logic languages: "residuation" and "narrowing" (see [11] for a survey on functional logic programming).

Curry's operational semantics is based on a single computation model, firstly described in [13], which combines lazy reduction of expressions with a possibly non-deterministic binding of free variables occurring in expressions. Thus, purely functional programming, purely logic programming, and concurrent (logic) programming are obtained as particular restrictions of this model. Moreover, due to the use of an integrated functional logic language, one can choose the best of the two worlds in application programs. For instance, input/output (implemented in logic languages by side effects) can be handled with the monadic I/O concept [28] in a declarative way. Similarly, many other impure features of Prolog (e.g., arithmetic, cut) can be avoided by the use of functions.

Beyond this computation model, Curry provides higher-order functions, a parametrically polymorphic type system, modules, monadic I/O, a connection to external functions, and primitives to encapsulate non-deterministic computations (committed choice, encapsulated search). To provide the full power of logic programming, (equational) constraints can be used in conditions of function definitions. Such basic constraints can be combined to more complex constraint structures by a concurrent conjunction operator which evaluates constraints concurrently. The concurrent conjunction of constraints is useful to reduce the search space with passive constraints or to model concurrent objects as functions synchronizing on a stream of messages.

Since Curry is an amalgamation of functional and logic programming languages, it can also serve as a common basis to unify research efforts and to boost declarative programming in general. Actually, Curry has been successfully applied to teach functional and logic programming techniques in a single course without switching between different programming languages [12].

In order to provide a portable implementation of Curry, we have developed an abstract machine for executing Curry programs. This abstract machine can be implemented in various languages but we use it as a link to compile Curry programs into Java [32].[1] The instructions of the abstract machine implement the pattern matching and control flow of a Curry program and can be executed by a small emulator.

Due to the use of Java in our implementation, compiled Curry programs are portable between different hardware architectures and operating systems. Furthermore, we exploit the concurrent features of Java in two ways in our implementation. *Don't know* non-deterministic computations (corresponding to search in logic programming) are implemented by starting independent computation threads (this corresponds to OR-parallel implementations of logic languages). The *concurrent conjunction* of constraints is implemented by starting two threads evaluating these constraints. Such a thread suspends if it requires the value of a currently unbound variable (consumer of the variable), or if it tries to bind a variable to different values (non-deterministic binding to compute potential solutions) while its corresponding twin thread is already active. The latter case avoids the duplication of active processes for don't know non-deterministic computations which is also known as *stability* from AKL [20] and the Andorra principle [19]. This scheme implements coroutining features of current Prolog systems [25] as well as features of concurrent constraint languages [30].

In the next section, we review the basic computation model of Curry. The abstract machine to implement this computation model is described in Section 3. Section 4 presents some extensions of the basic computation model and the corresponding extensions of the implementation. The implementation of our abstract machine in Java or related languages is discussed in Section 5, and Section 6 contains the results of our implementation and the conclusions.

[1]A straightforward direct translation of defined functions into Java functions is not possible in a simple way due to the potential non-deterministic evaluation of Curry expressions which we want to implement as independent computations (e.g., by Java threads) rather than sequentially as in Prolog. Thus, standard techniques to translate (sequential) logic programs into procedures of imperative languages (e.g., [26]) cannot be applied here.

2 The Computation Model of Curry

This section provides an informal introduction to the computation model of Curry. A formal definition can be found in [13, 16].

The basic computational domain of Curry is, similarly to functional or logic languages, a set of *data terms* constructed from constants and data constructors. These are introduced through data type declarations like[2]

```
data Bool   = True  |  False
data Nat    = Z     |  S Nat
data List a = []    |  a : List a
```

`True` and `False` are the Boolean constants, `Z` and `S` are the zero value and the successor function to construct natural numbers,[3] and `[]` (empty list) and `:` (non-empty list) are the constructors for polymorphic lists (`a` is a type variable ranging over all types).

A *data term* is a well-formed expression containing variables, constants and data constructors, e.g., `(S Z)` or `[x,y]` (the latter stands for `x:y:[]`). *Functions* (or *predicates* in logic programming, but throughout this paper we consider predicates as Boolean functions for the sake of simplicity) operate on data terms. Their meaning is specified by *rules* of the form l `=` r where l has the form $f\, t_1 \ldots t_n$ with f being a function, $t_1, \ldots, t_n$ data terms and each variable occurs only once (expressions of this form are also called *patterns*), and r is a well-formed *expression* which may also contain function calls. A *conditional rule* has the form l `|` c `=` r, where the condition c is a constraint. A *constraint* is any expression of the built-in type `Constraint`. Primitive constraints are equations of the form t_1 `=:=` t_2 which can be combined into more complex constraints by the concurrent conjunction operator `&` (see below for more details). A conditional rule can be applied if its condition is satisfiable. A *head normal form* is a variable, a constant, or an expression of the form $C\, e_1 \ldots e_n$ where C is a data constructor. A *Curry program* is a set of data type declarations and rules.

[2]Curry has a Haskell-like syntax [17], i.e., (type) variables and function names start with lowercase letters and the names of type and data constructors start with an uppercase letter. Moreover, the application of f to e is denoted by juxtaposition ("$f\ e$").

[3]Curry has also built-in integer values and arithmetic functions (see Section 3.3). We use here the explicit definition of naturals only to provide some simple and self-contained examples.

Example 1 *Assume that the above data type declarations are given. Then the following rules define the addition and the predicate "less than or equal to" for natural numbers:*

```
add Z     y = y
add (S x) y = S(add x y)
```

```
leq Z     x     = True
leq (S x) Z     = False
leq (S x) (S y) = leq x y
```

The following rules define the concatenation of lists and the last element of a list:

```
conc []     ys = ys
conc (x:xs) ys = x :  conc xs ys

last xs | conc ys [x] =:= xs  = x  where x,ys free
```

If the equation "`conc ys [x] =:= xs`*" is solvable, then* `x` *is the last element of the list* `xs`*. The optional part "*`where x,ys free`*" in the last rule declares* `x` *and* `ys` *as extra-variables that do not occur in the left-hand side.*

From a functional point of view, we are interested in computing *values* of expressions, where a value does not contain function symbols (i.e., it is a data term) and should be equivalent (w.r.t. the program rules) to the initial expression. The value can be computed by applying rules from left to right. For instance, we compute the value of `add (S Z) (S Z)` by applying the rules for addition to this expression:

```
add (S Z) (S Z)   →   S(add Z (S Z))   →   S(S Z)
```

The difficulty in computing such values is to find a *normalizing* strategy which selects a reducible function call ("*redex*") such that a value is always computed (provided that it exists). Since it is well known that innermost reduction is not normalizing, Curry is based on a lazy (outermost) strategy. This also allows the computation with infinite data structures and provides more modularity by separating control aspects [18].

A subtle point in the definition of a lazy evaluation strategy in combination with pattern matching is the selection of the "right" outermost redex. For instance, consider the rules of Example 1 together with the rule `f = f`. Then the expression "`leq (add Z Z) f`" has two outermost redexes, namely `(add Z Z)` and `f`. If we select the first one, we compute the value `True` after one further outermost reduction step. However, if we select the redex `f`, we run into an infinite reduction sequence instead of computing the value. Thus, it is important to know which outermost redex is selected. Since most lazy functional languages follow a left-to-right pattern-matching strategy and choose the leftmost outermost redex, Curry follows the same approach.[4] Thus, in order to evaluate the expression "`leq (add Z Z) f`", the first subterm `(add Z Z)` must be evaluated to head normal form (in this case: `Z`) since its value is required by all rules defining `leq` (we also call such an argument *demanded*). If the first subterm evaluates to an expression of the form `(S...)`, then the second subterm needs to be evaluated.

Sometimes there is no single argument in the rules' left-hand sides demanded by all rules. In particular, functions defined by rules with overlapping left-hand sides, like the "parallel-or"

```
  True ∨ x     = True
     x ∨ True  = True
 False ∨ False = False
```

have this property. For the expression $e_1 \vee e_2$, it is not clear which subterm must be evaluated first. Since our computation model must already include some kind of non-determinism in order to cover logic programming languages, we *non-deterministically evaluate* the first or the second argument, i.e., rules with overlapping left-hand sides cause don't know non-deterministic computations similarly to logic programming.[5]

Up to now, we have only considered functional computations where ground expressions are reduced to values. In logic languages, the initial expression (often an expression of Boolean type, called a *goal*) may contain free variables. A logic programming system should find values for these variables such that the goal is reducible to `True`. Fortunately, it requires only a slight extension of the reduction strategy to cover non-ground expressions and variable instantiation: if the value of a free variable is demanded by the left-hand sides of program rules in order to proceed the computation, the variable is non-deterministically bound to the different demanded values. For instance, if the function `f` is defined by the rules

```
f 0 = 2
f 1 = 3
```

[4]Since the exact evaluation strategy of Curry is specified using definitional trees [2, 13], Curry also supports more powerful evaluation strategies than current functional languages.

[5]Alternatively, one could evaluate both arguments in parallel [4], but such a parallel evaluation strategy requires more implementation effort.

(the integer numbers are considered as an infinite set of constants), then the expression "`f x`" with the free variable `x` is evaluated to `2` by binding `x` to `0`, or it is evaluated to `3` by binding `x` to `1`. In order to reflect not only the computed *value* (like in functional programming) but also the different variable bindings (*answers*, like in logic programming), the *computational domain* of Curry is a *disjunction* of answer/expression pairs where the disjunction reflects the (don't know) non-determinism. For instance, in the previous example the evaluation of "`f x`" is reflected by the following (non-deterministic) computation step:

$$\texttt{f x} \quad \to \quad \{\texttt{x=0}\}\,\texttt{2} \;\; \texttt{|} \;\; \{\texttt{x=1}\}\,\texttt{3}$$

(Here `|` denotes a disjunction, $\{\texttt{x=0}\}$ denotes a substitution (binding), and $\{\texttt{x=0}\}\texttt{2}$ represents a answer/expression pair.) A single *computation step* performs a reduction in exactly one unsolved expression of a disjunction. This reduction may yield a single new expression (*deterministic step*) or a disjunction of new expressions together with the corresponding bindings (*non-deterministic step*). For inductively sequential programs [2] (these are, roughly speaking, function definitions without overlapping left-hand sides), this strategy, called *needed narrowing* [3], computes the shortest possible successful derivations (if common subterms are shared) and a minimal set of solutions, and it is fully deterministic if free variables do not occur.[6]

Note that Curry is not based on a backtracking strategy. Backtracking is one possible (but unfair) implementation of disjunctions. However, the concrete evaluation order is not important for the computed results, since Curry has no side effects. Because there is no need to ensure a sequential backtracking semantics as in Prolog (e.g., unlike in the ACE parallel Prolog system [29]), we will implement disjunctive computations by independent computation threads. However, this implementation issue is less important than in Prolog since most parts of larger computations are purely deterministic due to the use of a functional coding style in Curry. In fact, there is an extension of this base language where don't know non-deterministic computations can be encapsulated and controlled by the programmer [15]. The abstract machine presented below is also an appropriate basis to implement such a flexible kind of search control.

In functional logic programs, it is necessary to solve equations between

[6]These properties also shows some of the advantages of integrating functions into logic programs, since similar properties for purely logic programs are not known.

expressions containing defined functions. For instance, to evaluate the expression "`last [1,2]`" w.r.t. the declarations in Example 1, the equation "`conc ys [x] =:= [1,2]`" has to be solved. This can be done by evaluating the equation to `[x1,x]=:=[1,2]` (i.e., `ys` is bound to `[x1]` and we omit the other alternatives in the disjunction) and unifying both sides of the resulting equation which yields the binding of `x` to `2`. In general, an *equation* or *equational constraint* e_1 `=:=` e_2 is satisfied if both sides e_1 and e_2 are reducible to the same data term. As a consequence, if both sides are undefined (non-terminating), then the equality does not hold.[7] Operationally, an equational constraint e_1 `=:=` e_2 is solved by evaluating e_1 and e_2 to unifiable data terms where the lazy evaluation of the expressions is interleaved with the binding of variables to constructor terms [23]. Thus, an equational constraint e_1 `=:=` e_2 without occurrences of defined functions has the same meaning (unification) as in Prolog. The basic kernel of Curry only provides equations e_1 `=:=` e_2 between expressions as constraints. Since it is conceptually fairly easy to add other constraint structures, future extensions of Curry will provide richer constraint systems to support constraint logic programming applications. Note that constraints are solved when they appear at the top level or in conditions of program rules (cf. Section 4) in order to apply this rule.

The use of functions together with lazy evaluation and demanded instantiation of free variables can reduce the number of non-deterministic choices compared to purely logic programming. On the other hand, it is also known that the guessing of free variables should not be applied to all functions, since some functions (in particular, functions defined on recursive data structures) may not terminate if demanded arguments are unknown. Moreover, many logic languages provide flexible selection rules (coroutining, i.e., concurrent computations based on the synchronization on free variables). To support such features, Curry provides the *suspension of function calls* if a demanded argument is not instantiated. Such functions are called *rigid* in contrast to *flexible* functions which instantiate their arguments if it is necessary to proceed their evaluation. As a default in Curry (which can be easily changed), non-constraint functions are marked as rigid and constraints (i.e., functions with result type `Constraint`) are marked as flexible. Thus, purely logic programs (where each predicate is represented as a constraint function) be-

[7]This notion of equality is also known as *strict equality* [9, 24] and is the only reasonable notion of equality in the presence of non-terminating functions.

have as in Prolog, and purely functional programs are executed as in lazy functional languages like Haskell.

If function calls may suspend, we need a mechanism to specify concurrent computations. For this purpose, constraints can be combined with the *concurrent conjunction* operator `&`. If c_1 and c_2 are constraints, then $c_1 \,\&\, c_2$ is a constraint which is evaluated by solving c_1 and c_2 concurrently. In a sequential implementation, $c_1 \,\&\, c_2$ can be evaluated by an attempt to solve c_1. If the evaluation of c_1 suspends, an evaluation step is applied to c_2. If a variable responsible to the suspension of c_1 was bound during the last step, the left expression will be evaluated in the subsequent step. In this way we obtain a concurrent behavior with an interleaving semantics. In the concurrent implementation presented in this paper, we assign different threads to the evaluation of c_1 and c_2. These computation threads synchronize on the bindings of common variables.

This fairly simple model for concurrent computations is able to cover applications of Prolog systems with coroutining [25]. For instance, if `gen` is a constraint which instantiates its argument with potential solutions (i.e., `gen` is flexible) and `test` checks whether the argument is a correct solution (i.e., `test` is rigid), then a constraint like "`test x & gen x`" specifies a "test-and-generate" solution where the test is activated as soon as its argument is sufficiently instantiated.

It is also interesting to note that this model is able to cover recent developments in parallel functional computation models like Eden [6] or Goffin [7]. For instance, a constraint of the form

```
x =:= f t1  &  y =:= g t2  &  z =:= h x y
```

specifies a potentially concurrent computation of the functions `f`, `g` and `h` where the function `h` can proceed its computation only if the arguments have been bound by evaluating the expressions `f t1` and `g t2` (provided that `h` is a rigid function).

The advantage of Curry's computation model is the clear separation between sequential and concurrent parts. Sequential computations, which form the basic units of a program, can be expressed as usual functional (logic) programs, and they are composed to concurrent computation units via concurrent conjunctions of constraints. Since constraints can be passed as arguments or results of functions (like any other data object or function), it is possible to specify general operators to create flexible communication architectures similarly to Goffin [7]. Thus, the same abstraction facilities can be used for sequential as well as concurrent programming. On the other hand, the clear separation between sequential and concurrent computations supports the use of efficient and optimal evaluation strategies for the sequential parts, where similar techniques for the concurrent parts are not available. This is in contrast to other, more fine-grained concurrent computation models like AKL [20], CCP [30], or Oz [31].

3 An Abstract Machine for the Execution of Curry Programs

In this section we present an abstract machine for the execution of Curry programs. We describe the implementation of data terms, defined functions, and the computational model of Curry (variable bindings, non-determinism, concurrency). The machine and its data structures are described in an object-oriented style which supports a typed and more structured specification in contrast to other presentations of abstract machines (e.g., [36]). We use Java [32] as a language for the concrete description of the machine. This is due to the fact that we also sketch an implementation of this machine in Java. However, the basic structure of the abstract machine can also be used in implementations of Curry in languages different from Java. This point will be further discussed in Section 5.

The main motivation for using Java is to provide a portable implementation of Curry and its concurrency features. Java is an object-oriented programming language where the compiler transforms source code not into native machine code but into code (the *bytecode*) for the Java Virtual Machine [21], abbreviated JVM, an abstract machine dedicated to execute Java programs. The JVM code is independent of a particular hardware. Therefore, a compiled Java program can run on different hardware architectures provided that a JVM implementation is available on this hardware. The JVM interprets the compiled bytecode and manages a heap where all objects constructed by the program reside. It is not necessary to call a destructor for unused objects, since the memory of unused objects is automatically reclaimed by the garbage collector of the JVM. By compiling Curry programs into Java programs and further down to JVM code, a compiled Curry application is executable on any computer with an installed JVM (e.g., a WWW browser). This is in contrast to the use of other languages like C or C++ where the programs must be recompiled on different architectures. Another advantage of using Java is its built-in support for automatic memory management and concurrent (thread-based) programming. This has largely simplified the implementation task for Curry. On the other hand, due to the use of Java, it is also evident that this implementation cannot compete with highly optimized implementations of Prolog or Haskell, since the hardware-independence and security mechanisms of the JVM causes a significant overhead in comparison to native code.

3.1 Representation of Data Terms and Functions

As mentioned above, we provide an object-oriented description of the structure of our abstract machine for compiling Curry programs. Therefore, dynamically created data terms and expressions are represented as objects of particular classes. We show the definition of these classes in Java, but they can be similarly implemented in C++, in which case an additional garbage collector is necessary.

Expressions or terms can contain constructors, functions as well as free variables. Therefore, we define an abstract class[8] `Term` which is a superclass for all possible objects occurring in expressions:

```
abstract public class Term {
  abstract public String toString();  // format this term
}
```

Hence all terms have a method `toString` (which must be redefined in the concrete subclasses) to produce a readable string representation of this term, e.g., for debugging or to print computed results.

Since the implementation of free variables is described below (Section 3.2), we show here only the implementation of terms headed by constructors or function symbols. The common property of such terms is the fact that they have a distinct number of arguments. Thus, we extract this property in an abstract class `NodeTerm` representing terms with arguments. Since the *maximal* number of arguments for each constructor and function is known at

[8]An *abstract class* is a class for which no instance can be created. Thus, abstract classes are used to describe common properties of (concrete) object types which are declared as subtypes of this abstract class. This has the advantage that the common properties need not be multiply defined in the concrete subclasses.

compile time (since Curry is a typed language), we can store all arguments in a local array:

```
abstract public class NodeTerm extends Term {
  public int nargs=0; // current number of arguments
  private Term [] args; // reference to arguments

  abstract public int getArity(); // max. number of args
  public final Term getArg(int n) { return args[n]; }
  public final void setArg(int n, Term t) { args[n]=t; }
  public final void addArg(Term t) { setArg(nargs++,t); }
  ...
}
```

Similarly to `Term`, the class `NodeTerm` is abstract since it is not reasonable to have any instances of this class. However, it provides the necessary functionality (methods `getArg`, `setArg`, `addArg`) for manipulating the current arguments of an expression. The method `getArity` returns the maximal number of arguments of a constructor or function and will be defined in the subclasses of `NodeTerm`. The values of `nargs` and `getArity()` are different in case of a partial application which can occur in higher-order programming (see Section 4.1).

The subclasses of `NodeTerm` are the classes `Constructor` and `Function` which represents constructor-headed expressions and function calls, respectively. In order to implement the pattern matching efficiently (see below), we assign to each constructor a unique index, i.e., for each data type occurring in a Curry program, all constructors are simply enumerated starting from 0. Thus, each instance of `Constructor` has in particular a method `getIndex` to access the index of this constructor. Therefore, the class `Constructor` has the following structure:

```
public class Constructor extends NodeTerm {
  private ConsDesc cd; // infos about name and type
  public int getIndex() {...}
  ...
}
```

In contrast to constructors, functions are evaluable. Thus, each function object has a component `code` containing the instruction sequence to be executed if this function call must be evaluated:

```
abstract public class Function extends NodeTerm {
  private FuncDesc fd; // infos about name and type
  static Command code[];
  Term result;
  public Command getCommand(int index) {
    return code[index];
  }
  ...
}
```

Due to the `static` declaration, the code is represented only once for all instances. The single commands of a function (see below) can be accessed by the method `getCommand`. The purpose of a Curry compiler is to translate each function definition into a definition of a subclass of `Function`. For instance, a function named `add` is translated into the class `Function_add` which contains in the component `code` the corresponding instructions of the abstract Curry machine, which will be described below.

The component `result`[9] of each function object contains the result of this call when this function is evaluated. This is necessary to implement laziness via sharing. A simple example shows why we need this component. Consider the declaration of the function `double`

```
double x = add x x
```

and the goal `double(add Z (S Z))`. If we evaluate this goal in a naive way (i.e., without sharing identical subterms), we evaluate the argument `(add Z (S Z))` twice (marked by underlining in the following derivation):

$$\begin{array}{lll}
\texttt{double(add Z (S Z))} & \rightarrow & \texttt{add }\underline{\texttt{(add Z (S Z))}}\texttt{ (add Z (S Z))} \\
 & \rightarrow & \texttt{add (S Z) }\underline{\texttt{(add Z (S Z))}} \\
 & \rightarrow & \texttt{add (S Z) (S Z)} \\
 & \rightarrow & \texttt{S(add Z (S Z))} \\
 & \rightarrow & \texttt{S(S Z)}
\end{array}$$

To avoid such redundant computations, lazy languages use term sharing to represent identical subterms. Furthermore, a function call is replaced by its result in order to avoid further evaluations of this function. The replacement

[9]The definition of this component will later be modified to handle non-deterministic computations.

of a function call by its result corresponds to transforming an instance of the class `Function` into an instance of the class `Constructor`, which is usually not possible in object-oriented languages. Therefore, we need another mechanism to store the result of a function evaluation in the corresponding term structure and a technique to detect that a function call has been already evaluated. This is done by an indirection through the `result` component of the `Function` class (as in [34]). A `result` value of `null` means that the function term has not been yet evaluated, otherwise it refers to the corresponding result term.

In the following, we will show the principles of our abstract machine by discussing the implementation of the simple function `add` of Example 1. For instance, a data term like `(S Z)` is represented by two instances of the class `Constructor` where the instance for the constructor `S` refers to the instance representing `Z`.

The creation of instances of these classes is performed by the Curry run-time system. As an example, consider that the user enters the expression "`add (S Z) Z`" to be evaluated. Then the run-time system performs the following operations:

1. An instance of the class `Function_add` is created that represents the initial expression. This instance refers to representations of the actual arguments `(S Z)` and `Z` which are also created as instances of the class `Constructor`.

2. The abstract Curry machine is started. Since the entered goal is the function `add`, the machine starts the execution of the code stored in the component `code` of the class `Function_add`.

Now we discuss the code instructions in more detail. The operational model of Curry requires that the code of the function `add` should do something like this:

```
if <1st argument is a function>
   <evaluate the function (to head normal form)>
if <1st argument is a constructor $C$>
   switch on $C$:
     case 'Z':
       <return argument 2 as result>
     case 'S':
       T = <first argument of this constructor 'S'>
       <return the expression 'S(add T <argument 2>)' as result>
```

Note that the lazy evaluation of expressions requires the creation of new, partially evaluated expressions like "`S(add T <argument 2>)`" in this example. Thus, the abstract Curry machine must be able to execute instructions

- to test the type of arguments (function or constructor),
- to construct new expressions (the right-hand sides of rules), and
- to change the execution path (e.g., call functions and return results).

To perform these tasks, the machine has the following *registers* and *data areas*:

`AReg`$_0$, `AReg`$_1$, `...`: These are the argument registers that hold the actual arguments when a function is called.

`TReg`$_0$, `TReg`$_1$, `...`: These are auxiliary registers for pattern matching, i.e., to get access to the arguments of constructor terms (see "`T`" in the pseudo code above).

`PC`: A program counter pointing to the next instruction. This program counter consists of two components: a reference to an instance of some `Function` subclass (the current function to be evaluated) and an index to a command in the `code` component of this instance (the next command to be executed). In a more efficient implementation, one could store the code for all functions in one memory area (as done in most implementations of abstract machines) so that the program counter consists of only one component (an address into this code area) which speeds up the access to the next instruction. On the other hand, the distributed storing of code in our approach has the advantage that it enables a dynamic loading of code. In fact, the Java run-time system loads only the code of those classes that are actually needed.

`TermStack`: This stack is mainly used to create new terms as needed by the right-hand sides of function rules. The instruction set of the machine contains commands to push references to terms onto this stack and commands to create new constructor/function terms using the contents of the stack as arguments for those new terms. Later we will see that we can avoid the use of the stack in most cases.

`CallStack`: This stack tracks function calls. Every function call creates a new entry on the top of the stack containing the following information:

1. A reference to the function call to be evaluated.
2. A copy of the relevant argument registers.
3. A reference to the command that will be executed when the machine returns from the function call. This is often called the *return address* in compiler construction [1] or the *continuation pointer* in the WAM [36].
4. The index of the argument register where the result of the evaluation of the function call should be stored.

In fact, an entry in the call stack represents the state of the machine before it called the function. Thus, the machine state can be restored from the top entry when the machine returns from the function call.

In our object-oriented description, the single commands of the abstract machine are represented by instances of classes that are derived from the abstract class `Command`:

```
abstract public class Command {
  abstract void execute(...);
}
```

Thus, a command is executed by calling its `execute` method. A machine program (i.e., a sequence of commands) is implemented as an array of `Command` objects (stored in the component `code` of functions). The complete code of the function `add` looks like this:

```
public class Function_add extends Function {
  private static final Command code[]={
  // Argument 1 is in AReg$_0$, Argument 2 in AReg$_1$
    new CmdSwitchOnType(0,11), // AReg$_0$ function -> line 11
  // code for case 'isConstructor'
    new CmdSwitchOnCons(0,{2,4}), // AReg$_0$:  Z -> line 2
                                  //          S -> line 4
  // code for rule 1 (this is line 2):
    new CmdPushReg(1),
    new CmdReturn(), // return AReg$_1$ as result
  // code for rule 2 (this is line 4):
    new CmdLoadArgs(0),  // get arg. of constructor:
    new CmdLoadReg(0,0), // S x -> x
    new CmdPushReg(0),
    new CmdPushReg(1),
    new CmdPushFunc("add",2),
    new CmdPushCons("S",1),
    new CmdReturn(), // return S(add x y) as result
  // code for case 'isFunction' (this is line 11):
    new CmdCall(0,2,0), // execute function call in AReg0,
                  // save 2 registers and return to line 0
  };
}
```

The first command `CmdSwitchOnType` checks whether argument register `0` refers to a function. If this is the case, it jumps to line 11 (all addresses are indices in the command array) where the function is evaluated by `CmdCall(0,2,0)`, otherwise it proceeds with the next instruction. The instruction `CmdSwitchOnCons(0,{2,4})` is a case distinction on the value of the top constructor symbol in register 0: if it is the constructor `Z`, we jump to line 2, if it is the constructor `S`, we jump to line 4 (remember that we assign a unique index to each constructor of a data type; these are exactly the indices used in the jump table `{2,4}`). If the constructor is `Z`, the contents of argument register 1 (containing the second argument) is pushed on the term stack (`CmdPushReg(1)`) and we return with the top element of the term stack by the instruction `CmdReturn()`. The subsequent 7 instructions create the right-hand side of the second rule (`S(add x y)`) using the term stack: `CmdLoadArgs(0)` loads the arguments of the top constructor symbol of register `0` into the auxiliary registers. In this case, the single argument `x` is loaded into $\texttt{TReg}_0$. This argument is moved into register $\texttt{AReg}_0$ by `CmdLoadReg(0,0)`. Finally, the new expression is created using push instructions (e.g., `CmdPushFunc("add",2)` creates an `add`-headed term with the two top elements from the term stack as arguments).

This is only a brief description of the abstract machine but a more precise definition of the individual commands can be found in the appendix. This abstract machine is basically a reduction machine for the implementation of functional languages (like [22]) enhanced with the parameter passing mechanism and the pattern matching from the WAM [36].

Our current implementation additionally performs some optimizations. One of these optimizations consists in compiling right-hand sides of rules which do not contain control transfer instructions directly into Java code. This avoids the stepwise execution of the command sequence and the use of the term stack for the construction of terms. For instance, two successive push and pop instructions on the term stack, like "`PushReg(1); Return()`" in rule 1 of the function `add`, are replaced by a direct return with the contents of argument register 1 ("`machine.returnWith(machine.getReg(1))`", see also Appendix A.2.6). This optimization can be implemented by a new command named `CmdCustom` which allows the machine to call Java methods directly. Then we are able to replace the command sequences corresponding to the right-hand sides of rule 1 and 2 by `CmdCustom` commands calling the corresponding Java methods `rule1` and `rule2`:

```
public class Function_add extends Function {
  private static final Command code[]={
    new CmdSwitchOnType(0,4),
    new CmdSwitchOnCons(0,{2,3}),
    new CmdCustom(rule1),
    new CmdCustom(rule2),
    new CmdCall(0,2,0),
  };
  private static void rule1(Machine machine) {
    machine.returnWith(machine.getReg(1));
  }
  private static void rule2(Machine machine) {
    Constructor cons=(Constructor)machine.getReg(0);
    Constructor cons0=new CustomConstructor(consDesc0);
    Function func0=new Function_add();
    func0.addArg(cons.getArg(0));
    func0.addArg(machine.getReg(1));
    cons0.addArg(func0);
    machine.returnWith(cons0);
  }
  private static final ConsDesc consDesc0 =
                                Run.findConsDesc("S");
  ...
```

```
    }
```

The methods `rule1` and `rule2` can be interpreted as the inline expansion (together with some simple optimizations) of the corresponding commands in our first definition of `Function_add`. Note that passing methods as arguments is allowed only in the Java extension “Pizza” [27] which we used for our implementation. Pure Java programmers need to simulate this by passing objects containing these methods. The use of Pizza does not reduce the portability of our implementation since the Pizza extensions are compiled into Java and hence to the JVM.

In this section we have described only the implementation of the purely functional part of Curry. However, Curry also inherits from logic programming free variables and non-determinism. The implementation of these features is sketched in the following section.

3.2 Free Variables and Disjunctive Computations

Since Curry subsumes purely logic languages, we have to implement free variables and their (non-deterministic) bindings. Free variables may be bound by pattern matching or by unification of an equational constraint. We describe only the binding by pattern matching, since the unification of data terms can be similarly implemented (iterative pattern matching). The main problem to bind a free variable by pattern matching is caused by the fact that a variable may be bound to different constructors leading to alternative solutions. We implement such non-deterministic bindings by starting several abstract Curry machines which evaluate the expressions with the different bindings of the variable.

Example 2 *Consider the rules for* `add` *of Example 1 and the evaluation of the expression “*`add y Z`*”. Since* `y` *must be bound to* `Z` or `(S x)` *(where* `x` *is a new free variable) to proceed the evaluation, we create* two *machines for the further evaluation: one machine works with the binding* {`y=Z`} *and the other with* {`y=(S x)`}.

Thus, the basic idea of our implementation of disjunctions is to have *one machine for each disjunction* occurring in a computation. Each machine is responsible to evaluate an expression as described in the previous section. Since Curry is not based on a backtracking strategy, there is no fixed order in the execution of these machines. In our current implementation described in this paper, these machines are executed as independent Java threads in a fair manner, but this is not essential. Our design is also intended as a basis for future extensions where the search strategy, i.e., the order to traverse the search tree representing disjunctive computations, can be specified by the programmer [15]. In this case, the order of executing the machines can be arbitrarily interleaved which requires that switching the execution between different machines and creating new machines should be efficiently performed.

There are different approaches to handle alternative bindings for a variable, like copying the complete environment for each machine, storing the bindings in tables, called binding arrays, or using hash tables for the efficient access to the binding of a variable w.r.t. a machine. All these techniques have different costs w.r.t. the time needed for machine creation or variable access (see, e.g., [10] for a discussion on different implementation technqiues). To ensure constant machine creation time and to exploit the object-oriented design of our implementation, we have decided to move the binding information into the variable objects. Thus, we give the variable objects the capability to determine their bindings rather than collect the variables and their bindings into a special data structure. For the implementation of alternative bindings, we assign to each machine a unique natural number (called *key*). Furthermore, each variable object contains a structure called *binding table* which is in fact a hashtable of type `Key` $\rightarrow$ `Term`, i.e., variables are objects of the following type:

```
public class Variable extends Term {
  String name; // variable name for debugging
               // and printing results
  BindingTable bindingtable = new BindingTable();
  ...
}
```

The machine can use its key to “ask” the variable object whether the binding table contains bindings for the asking machine. When the machine wants to bind the variable, its key is used to insert a new entry in the binding table.

There is still one problem: consider a machine A which binds a variable V, then causes a non-deterministic computation which creates a new machine B, and finally binds a variable W. The machine B may not only access bindings created by itself but access bindings created by the machine A, too, since A and B have the same “prehistory”. How can machine B differ between the binding of V which is part of their common prehistory and the binding of W (which is not)?

To solve this problem, we add *timestamps* to every machine and to every entry in a binding table. The timestamps are handled by two rules:

1. Machines have two timestamps, called "*current timestamp*" and "*birth timestamp*". When a new machine is created, its current timestamp and its birth timestamp are set to the value of the current timestamp of the creator machine. Then the current timestamp of the creator is incremented by one.

2. If a machine creates an entry in a binding table, it stores its key as well as its current timestamp in the entry.

When a machine B wants to access a variable U, two cases may occur:

1. The binding table of U contains an entry for B: this means that U is bound in the context of B.

2. The binding table contains no entry for B: we have to restart the lookup with the *key of the creator machine* of B (see our example above).

   If this extended search did not find an entry, we can stop the search: U is really not bound (in the context of B). Otherwise, we have to check for the found `entry` that

   `entry.timeStamp` $\leq$ B`.birthTimeStamp`

   This means that B inherits this binding from its creator. In this case, we can avoid the lookup for the future by creating a new entry for B now (but with the same timestamp `entry.timeStamp`).

The described technique guarantees constant machine creation time since binding tables are not copied. If we initialize the binding table big enough, we have in most cases also a constant variable access time (except for the first access).

Due to different variable bindings in the various running machines, a function call may also be evaluated to different results in these machines. Therefore, we use the same binding technique also for storing the result of function calls, i.e., the component `result` of the `Function` class refers to a binding table instead of a simple term.

We describe the new commands of our abstract machine to deal with non-determinism and variable bindings by completing the code of our `add` example of the previous section. In order to extend the code in the class `Function_add` to include the possible bindings of free variables, we replace the first command (`SwitchOnType`) by the instruction

```
SwitchOnType(0,11,12) // AReg0 contains function -> line 11
                      // AReg0 contains variable -> line 12
```

Thus, the command `SwitchOnType(`n`,`$l1$`,`$l2$`)` inspects the contents of argument register n and jumps to line $l1$ if it is a function call, and to line $l2$ if it is a free variable. Otherwise, it proceeds with the next instruction.

Furthermore, we append the following code sequence (case "*is a variable*") at the end of the previous code for `add` (here we show the generated code instead of the creation commands "`new Cmd...`"):

```
// handling of a free variable as first argument
// (this is line 12):
  NewChoice(16,2), // start copy of this machine at line 16
// choice 1:
  PushCons("Z",0),
  SimpleBind(0),   // bind variable in register 0 to 'Z'
  Jump(2),
// choice 2 (this is line 16):
  PushVar(),       // create new variable 'x'
  PushCons("S",1),
  SimpleBind(0),   // bind variable in register 0 to 'S x'
  Jump(4)
```

If the first argument of a call to `add` is a free variable, the current computation thread (machine) is duplicated. One machine binds the free variable to the constructor `Z` and proceeds with the first rule, while the other machine binds the variable to `(S x)` (where `x` is a newly created variable) and proceeds with the second rule.

3.3 Concurrent Execution of Constraints

As explained in Section 2, Curry also supports the suspension of function evaluations that are not sufficiently instantiated. As known from logic programming with coroutining [25], this feature is quite useful to avoid an unrestricted generation of infinite solution sets and also supports a concurrent programming style like in concurrent constraint/logic programming languages. Moreover, the suspension of function calls is essential to connect *external functions* in a clean way [5]. For instance, the addition on integers is conceptually defined by an infinite set of rules:

```
0+0 = 0
0+1 = 1
...
2+3 = 5
...
```

These rules are not explicitly available in the program but this addition is implemented by some external operation. Therefore, an addition with unknown values, like `x+3=:=y` must be delayed until the arguments are known. Since Curry supports the concurrent execution of constraints by the concurrent conjunction operator `&`, a goal like "`x+3=:=y & x=:=2*3`" is executed by suspending the first constraint, evaluating the second constraint which binds `x` to `6`, and activating the first constraint which evaluates `6+3` and binds `y` to `9`.

It is reasonable to implement the concurrent conjunction of constraints by concurrently working machines, where the machines evaluate different constraints. The synchronization of these machines is organized through the binding of common variables. This has the consequence that the run-time system must create new machines not only for OR-parallel computations but also for AND-parallel computations.

To implement this feature, our previous concept of a strong connection between a machine and the expression/goal to be evaluated must be replaced by a new data structure. This structure (the so-called *computation structure*) contains the AND-tree representing the structure of concurrent conjunctions. This tree organizes the AND-parallel evaluation of a goal:

- The leaves represent working machines evaluating a constraint.
- An evaluation of a concurrent conjunction $c_1 \,\&\, c_2$ replaces the calling leaf (i.e., the machine which attempts to evaluate this constraint) by an AND-node with two sons that represent the constraints c_1 and c_2, respectively.

- If the two sons of an AND-node terminate successfully (i.e., the constraints are solved), this AND-node is replaced by a leaf which continues the evaluation of the expression containing this conjunction.
- A failure of one leaf results in a failure of the entire computation structure, since the entire constraint is unsolvable.
- The computation structure terminates successfully only if all leaves have terminated successfully.

Since function calls may suspend if free variables are passed as arguments (e.g., `x+3`), an AND-tree may contain two types of leaves:

1. *Active leaves* represent running machines.
2. *Suspended leaves* represent machines that are waiting on a variable binding in order to proceed their computation. Since this binding can only be done by an active leaf, the lack of any active machine in the tree will cause a failure of the entire computation structure.

Since the variable objects with their binding tables are shared by all machines of the computation structure, it is fairly simple to organize the access to a variable in a synchronized way, which is supported in Java by the monitor concept. Only one machine at a time can access and bind the value of a particular variable. A problem occurs if one machine wants to bind a variable to different values (OR-parallelism). The two extreme solutions to this problem (among many others, see, e.g., [10]) are the complete copying of the entire computation structure (which corresponds to full OR-parallelism) or to forbid the non-deterministic binding inside concurrent computations. We prefer a middle course and adopt a solution which is related to *stability* in AKL [20]:

- Leaves which want to bind a variable deterministically are allowed to do it immediately.
- Non-deterministic binding will cause the leaf to suspend. This leaf can only be reactivated
  - if there is no other active leaf, or
  - if another active leaf has bound the variable.

This can avoid unnecessary duplications of active machines and prefers deterministic parts of the computation, similarly to the Andorra computation model [19]. If all leaves in a computation structure are suspended but there are several leaves waiting to bind a variable non-deterministically, then only one of them (e.g., the leaf corresponding to the leftmost constraint with a non-deterministic binding attempt) is reactivated.

A small example helps to see how this works. Our program consists of one constraint function `digit` defined by the following rules, where `success` is a predefined constraint which is always satisfiable:

```
digit 0 = success
⋮
digit 9 = success
```

This (flexible) function acts in goals like

```
x+x=:=y & x*x=:=y & digit x
```

as a generator for values of `x`. The bindings produced by this generator are consumed by the (rigid) arithmetic functions `+` and `*`:

1. When the goal is started, a computation structure with three leaves for the constraints `x+x=:=y`, `x*x=:=y` and `digit x` is created.
2. If the machine evaluating "`digit x`" attempts to bind `x` in a non-deterministic way, it suspends as long as there are other active leaves.
3. The evaluations of the functions `+` and `*` suspend since all arguments are uninstantiated.
4. Since all consumers are now suspended, the machine for "`digit x`" is reactivated and creates bindings for the variable `x`.
5. The leaves which represent the consumers receive a message that the variable has been bound. This cause the reactivation of the machines.

Note that the entire computation structure must be copied before reactivating the machines in case of a non-deterministic binding.

In order to avoid the creation of too many concurrently working machines, we can introduce a simple optimization in the processing of the concurrent conjunction operator. Instead of immediately evaluating both constraints of an AND-node by two machines, the machine for evaluating the second constraint is only (concurrently) activated if the evaluation of the first constraint suspends, otherwise (i.e., if the first constraint can be fully evaluated without suspension), the second constraint is evaluated after the evaluation of the first one. This results in a sequential left-to-right execution of constraints (similarly to Prolog) that will only be broken when a constraint suspends.

3.4 Global Synchronization

The user interface of the run-time system consists of a main thread waiting for user-input. When an expression or goal has been entered, the main thread creates a new computation structure for this expression (containing initially only one leaf) and starts its execution.

A terminating computation structure sends a message to the main thread. This message may include a solution for the goal or simply signals a failed computation. In the former case, the main thread writes the solution on the terminal and re-enters into the message-loop, ready to receive further messages.

Note that—unlike the backtracking mechanism of Prolog—an infinite computation can not inhibit the output of other solutions, since the OR-parallel computation paths are executed as independent threads in our Java-based implementation.

4 Extensions of the Basic Computation Model

4.1 Higher-order Functions

Higher-order functions have been shown to be very useful to structure programs and write reusable software [18]. Although the computation model described so far includes only first-order functions, higher-order features can be implemented by providing a (first-order) definition of the application function (as shown by Warren [35] for logic programming). Curry supports the higher-order features of current functional languages (partial function applications, lambda abstractions) by this technique, where the rules for the application function are implicitly defined. In particular, function application is *rigid* in the first argument, i.e., an application is delayed until the function to be applied is known (this avoids the expensive and operationally complex synthesis of functions by higher-order unification).

To recognize partially applied expressions, every constructor or function call (i.e., every instance of a subclass of `NodeTerm`) stores the arity of the constructor or function *and* the number of arguments currently bound to the constructor/function object (method `getArity` and field `nargs` of class `NodeTerm`, see Section 3.1). If a partial application (i.e., an object whose number of arguments is less than its arity) is applied to an argument, the new argument is simply added to the object and the argument counter is incremented.[10] If a function call has now the same number of arguments as its arity, it is evaluated.

For instance, consider the function `map` which applies a function to all elements of a list:

```
map f []     = []
map f (x:xs) = f x : map f xs

inc x = x+1
```

Then the expression "`map inc [0,2,1]`" evaluates to the list `[1,3,2]`. In the first step of this evaluation, the expression is reduced to "`inc 0 : map inc [2,1]`". The application of `inc` to `0` adds the constructor `0` as a new argument to the partial function call `inc`. Since the arity of `inc` is `1`, this function call has now the maximal number of arguments and is evaluated to `1`.

4.2 Conditional Rules

Conditional rules, in particular with *extra variables* (i.e., variables not occurring in the left-hand side) in conditions, are one of the essential features to provide the full power of logic programming. Our implementation can be easily extended to conditional rules following the approach taken in Babel [24]: consider a conditional rule "l | c = r" as syntactic sugar for the rule $l = (c \Rightarrow r)$, where the right-hand side is a *guarded expression*. The operational meaning of a guarded expression "$c \Rightarrow r$" can be defined by the rule

```
(success ⇒ x) = x
```

[10]Note that the machine has to make a copy of the function object since it may be shared with other expressions.

(for this purpose, `success` is considered as a constructor). Thus, a guarded expression is evaluated by an attempt to solve the constraint. If this is successful (i.e., leads to the satisfiable constraint `success`), the guarded expression is replaced by the right-hand side r of the conditional rule.

4.3 Committed Choice

To support the usual concurrent (logic) programming techniques, Curry has also a committed choice construct to express a don't care choice between different alternatives. A committed choice expression like

```
choice c1 -> e1
       ...
       cn -> en
```

(where $c_1, \ldots, c_n$ are constraints and $e_1, \ldots, e_n$ are expressions of the same type) instructs the machine to make a don't care choice between the different execution paths $e_1, \ldots, e_n$ provided that the corresponding contraint is satisfiable. To accomplish this task, all guards $c_1, \ldots, c_n$ are evaluated in an OR-parallel manner and the machine chooses the expression whose corresponding constraint terminates successfully. If more then one constraint is successful, the machine has to choose non-deterministically one of them. If all constraints fail, the entire expression fails. If all constraints suspend, the entire expression suspends. Unlike the (don't know) non-determinism discussed in the sections above, this choice is *don't care*: there is no alternative if the choosen expression does not lead to a solution.

Since the machine does not know which constraint will be choosen, it is important that the constraints must not change their execution environment: a constraint c_i is only allowed to bind variables that are local in c_i (and the corresponding expression e_i). Additionally, if a constraint produces non-determinism, all new copies of this constraint are handled like the other members of the committed choice (i.e., the constraints in the committed choice are handled as *deep guards*).

The correct implementation of this behavior requires the distinction between local and global variables. To support this, we extend the computation structures and the variable objects by a natural number called "*level*":

- Machines that are part of a committed choice have got the level incremented by one of the machine that has initiated the committed choice.

- Machines that are copies of another machine inherit the level of the creating machine.
- The *level of a variable* is set to the level of the machine that creates the variable.

Now we are able to control variable accesses of a machine with level l:

- Variables with a level less than l are global to this machine and read-only. If the machine tries to bind such a variable, it suspends immediately.
- If the machine can not find a binding entry for a variable using its key, it restarts the lookup using the key of the corresponding machine with level $l-1$ (i.e., the machine that controls the enclosing committed choice).
- If a constraint of the committed choice terminates successfully, the controlling machine (evaluating the committed choice) inherits all bindings made by this constraint and continues with the evaluation of the corresponding expression.

4.4 Input/Output

Curry uses the monadic I/O concept from functional programming [33] to handle input/output in a declarative way. With this concept, an interactive program is considered as a function computing a sequence of actions which are applied to the outside *world*. Each action changes the state of the world and possibly returns a result (e.g., a character read from the terminal). The important point is that the world is not directly accessible to the programmer — she/he can only create and sequentially compose actions on the world. Due to the sequential composition of actions, only one version of the world is available at a time and, therefore, it is not necessary to store the current state of the world during the computation.

A problem might occur if an interactive program produces a disjunction as a result. Since the state of the world cannot be copied (note that the world contains at least the complete file system or the complete Internet in web applications), a disjunction of actions must be avoided.[11] Therefore, the programmer must encapsulate all possible disjunctive computations between I/O

[11]Actually, it is a run-time error if such a situation occurs.

operations, either by the committed choice described in Section 4.3 (where all disjunctive computations in the guards are encapsulated) or by the explicit encapsulation of search as described in [15]. As a consequence, one can use in Curry the same implementation techniques for monadic I/O as in functional languages.

5 Implementation of the Abstract Machine

We have used Java as the description language for the overall structure of our abstract machine. Therefore, this description can be directly used for an implementation of our machine in Java. In this case we have to implement a compiler which translates each function definition of a Curry program into a class definition (a subclass of `Function`, see Section 3.1) that contains the commands of our abstract machine implementing this function definition. Each abstract machine (responsible for the evaluation of a single Curry expression) can be executed as a separate thread in Java, i.e., the structure of a machine could be defined as

```
public class Machine extends Thread {
  // the function processed by the machine:
  private Function pcFunction;
  // the next command of the processed function:
  private int pcIndex;
  ...   // definitions of further registers, term stack,
        // call stack etc.
}
```

Since the machines are executed concurrently by the Java run-time system, all changes to common resources, i.e., variables and their bindings, must be organized in a synchronized way. This is easily possible in Java by the monitor concept.

If we use another language than Java for the implementation of our abstract machine, more effort for the implementation of the run-time system is needed, but we have also more possibilities for optimization. For instance, if we use C++, we have to implement a garbage collector and the concurrent or interleaved execution of machines. On the other hand, we can optimize the machine's implementation at various places:

- Since C++ does not support the dynamic loading of code, there is no advantage of attaching the abstract machine code for each compiled function to individual classes. Thus, the code for all functions can be collected in one array so that the program counter is a simple index into this array (instead of the two component program counter described above and in Section 3.1).

- By implementing a specific garbage collector, one can obtain a better memory management compared to the use of a standard garbage collector. For instance, the use of the built-in garbage collector in our Java implementation requires the explicit setting of unused argument or temporary registers of a machine to `null` values, otherwise some garbage cannot be reclaimed.

- Using C++, we have access to some features of the underlying operating system. For instance, we may use the memory management unit to implement copy-on-write data structures (i.e., data structures that are shared between two processes but which are duplicated if one of the processes tries to write to it). This would help us to avoid structures like the binding table in variable objects or the result table in function objects.

6 Conclusions and Results

We have presented an abstract machine to compile programs of the multi-paradigm language Curry and sketched its implementation in Java. Thus, the abstract machine can also be seen as a link to compile Curry into Java to obtain a portable implementation of Curry programs. Since Curry contains features from functional, logic, and concurrent programming, the implementation also reflects them: functional computations are implemented by an elementary stack-based machine, and the concurrent features of Java are exploited to implement the logic (OR-parallelism) and concurrent (AND-parallelism) elements of Curry.

The fact that Curry does not depend on a backtracking semantics like Prolog (since Curry has no side effects) has simplified the structure of the implementation. The user does not need to take any precautions when he wants to use parallelism in his program. The explicit notion of a concurrent conjunction results in a transparent AND-OR-parallelism that can be controlled with the high-level concept of constraints. For example, the evaluation of a goal like "`generate x & test x`" does not show any difference—neither in result nor in execution time—to the evaluation of "`test x & generate x`". Concerning this, the implementation is comparable with the Andorra model [19].

Since the run-time system of our implementation is written in Java, its speed is highly dependent on the efficiency of the JVM. First results of our implementation indicate that our implementation is much slower than highly optimized implementations of Prolog or Haskell (similar results have been reported by other projects to compile declarative languages in Java or the JVM, e.g., [34]). This is not due to the use of threads to implement AND/OR-parallelism, but it is caused by the fact that the JVM performs many run-time checks to ensure a reliable execution of Java programs. For instance, the (completely deterministic) execution of the classical "naive reverse" benchmark (note that it is executed by Curry in a (more costly) lazy manner in contrast to Prolog) is performed with approximately 14000 LIPS ("*L*ogical *I*nferences *P*er *S*econd", where a logical inference is here a reduction step with a rule) on a Linux-PC (Pentium II, 400 Mhz).[12] The use of an intermediate abstract machine in our implementation causes only a limited overhead: a direct translation of the functions occurring in the naive reverse example into Java procedures, which is possible due to absence of non-deterministic choices in this example, yields an efficiency improvement with a factor around 2.

In order to get some idea about the costs of threads in our Java-based implementation, we have compared a purely functional program with a similar program that makes extensive use of threads for implementing don't know non-deterministic choices:

```
linear1 n = if (n>0)==True then linear1 (n-1) else 0

linear2 0 = 0
linear2 n | (n>0)=:=True = linear2 (n-1)
```

[12]All runtimes are measured in real time, since Java does not support the access to the cpu time but only to the current time. Similarly, it is even more difficult to get some figures about the memory consumption since the standard profiling option provides only numbers about the totally used heap which does not differentiate between system objects and objects created by the user program. Thus, we cannot provide any numbers on the memory consumption.

The function `linear1` decrements its argument to zero in a fully deterministic way due to the explicit use of `if-then-else`. The function `linear2` simulates the sequential `if`-structure of `linear1` by two rules with overlapping left-hand sides which cause the creation of OR-parallelism.[13] This means that the goal "`linear2` n" creates $n + 2$ threads (unlike the goal "`linear1` n" which creates only one).

The execution times (in milliseconds) for different values of n are summarized in the following table:

n:	50	100	200	300	400	500
`linear1`:	17	35	69	104	137	172
`linear2`:	34	68	136	204	272	341
Ratio (`linear2`/`linear1`):	2.0	1.95	1.97	1.96	1.99	1.98

The result is as expected: The execution times for both functions are linear in n. Additionally, the function `linear1` is only around two times faster than `linear2`. This means that the current implementation of the JVM is able to create and handle Java threads very cheaply.

We have also tested the behavior of generating threads to implement the non-deterministic search (OR-parallelism) in combination with concurrent evaluation of constraints. For this purpose, we executed a highly non-deterministic program (a map coloring problem with 48 solutions). The program consists of a generator which may generate 256 different potential solutions and of a test function which identifies 48 of the possibilities as solutions. Combining these two functions to a generate-and-test system results in the creation of 256 threads during an execution time of 0.4 seconds. A test-and-generate system, where the test function suspends and acts as a passive constraint which is activated when it receives some input, creates only 124 threads (not all potential solutions are completely generated) but still consumes 0.25 seconds due to the higher degree of synchronisation between processes.

Due to use of Java threads for disjunctive and concurrent computations in our implementation, we have no influence on the scheduling of these threads since this is done by the Java run-time system. This limitation has also an advantage: If there is an implementation of the JVM available on multi-

[13]As explained in Section 2, rules with overlapping left-hand sides (i.e., if the left-hand sides are unifiable) are implemented as OR-parallel evaluations. In this example, one of the two generated threads always fails after pattern matching or condition checking.

processor architectures, which assigns threads to different processors, we immediately obtain a parallel implementation of our Curry programs without any change in our implementation.

Since Java is relatively new and Java compilers (in particular Just-in-Time-compilers) are currently not comparable to highly developed compilers like GNU-C, we can expect further efficiency improvements in the future. On the other hand, a good efficiency was not the main motivation to use Java. The concurrency features of Java together with its automatic memory management simplified our implementation. Moreover, the use of objects to represent data structures supports an easy connection of Curry with external functions directly implemented in Java.

In order to get an idea about the loss of efficiency due to the use of Java instead of C++, we have implemented the deterministic part of our abstract machine and its run-time system in C++. In this implementation, we have translated the classes describing the data structures and commands of our abstract machine as described in Section 3 into C++ in a straightforward way but already using some of the optimizations discussed in Section 5 (e.g., the code of all functions is stored in one array and the management of the heap is explicitly programmed). This implementation executes the "naive reverse" benchmark with approximately 740000 LIPS which is more than 50 times faster(!) than the Java-based implementation. This shows that the efficiency problem of our Java-based implementation is mainly due to the current JVM implementation.

The use of Java as an implementation language has—in spite of its efficiency—also other advantages: the ability of the JVM to load classes during run time allows the user to hold only those functions in memory that are actually needed (this feature has also been used by other projects like the Prolog-to-Java compiler JProlog [8]). For instance, the standard prelude for Curry programs contains many predefined functions which are available in all applications programs. In our implementation, the Java run-time system loads only the code of those functions which are actually evaluated. And last but not least: since the Java compiler produces bytecode for the JVM, the entire Curry run-time system is portable to other platforms: any computer with an installed JVM (e.g., a WWW-browser) is able to run an application written in Curry without recompilation of the source files.

Since Curry is intended to combine the paradigms of functional and logic programming, the application areas of Curry are similar to functional and logic languages (symbolic computations, knowledge-based systems, search problems, etc). Moreover, Curry is a good basis to teach functional and logic programming in a single course [12]. The Java-based implementation of Curry described in this paper has also been applied to solve problems where extensive search is needed (e.g., to implement a musical application where appropriate chords for the accompaniment of a given melody are generated [14]). The advantages of Curry for such applications is the smaller (demand-driven generated) search space due to the lazy operational semantics. For the future, we plan to integrate constraint solvers into Curry in order to apply it in applications from constraint logic programming.

Future extensions of this implementation will include methods to restrict and control the non-determinism by the encapsulation of search [15]. Furthermore, we will investigate program analysis methods to detect deterministic subcomputations and to transform lazy into eager evaluation. This would provide a method to translate parts of a Curry program directly into Java code which avoids the indirect execution by our abstract Curry machine.

Acknowledgements. The authors are grateful to the anonymous referees for their detailed comments on a previous version of this paper.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Mass., 1985.

[2] S. Antoy. Definitional trees. In *Proceedings of the 3rd International Conference on Algebraic and Logic Programming*, pages 143–157. Springer Lecture Notes in Computer Science 632, 1992.

[3] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Proceedings 21st ACM Symposium on Principles of Programming Languages*, pages 268–279. ACM Press, 1994.

[4] S. Antoy, R. Echahed, and M. Hanus. Parallel evaluation strategies for functional logic languages. In *Proceedings of the Fourteenth International Conference on Logic Programming (ICLP'97)*, pages 138–152. MIT Press, 1997.

[5] S. Bonnier and J. Maluszynski. Towards a clean amalgamation of logic programs with external procedures. In *Proceedings 5th Conference on*

Logic Programming & 5th Symposium on Logic Programming (Seattle), pages 311–326. MIT Press, 1988.

[6] S. Breitinger, R. Loogen, and Y. Ortega-Mallen. Concurrency in functional and logic programming. In *Fuji International Workshop on Functional and Logic Programming*. World Scientific Publ., 1995.

[7] M.M.T. Chakravarty, Y. Guo, M. Köhler, and H.C.R. Lock. Goffin - higher-order functions meet concurrent constraints. *Science of Computer Programming*, 30(1-2):157–199, 1998.

[8] B. Demoen and P. Tarau. JProlog. Available at `http://www.cs.kuleuven.ac.be/~bmd/PrologInJava/`, 1996.

[9] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel LEAF: A logic plus functional language. *Journal of Computer and System Sciences*, 42(2):139–185, 1991.

[10] G. Gupta. *Multiprocessor Execution of Logic Programs*. Kluwer Academic Publishers, 1994.

[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. Teaching functional and logic programming with a single computation model. In *Proceedings Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP'97)*, pages 335–350. Springer Lecture Notes in Computer Science 1292, 1997.

[13] M. Hanus. A unified computation model for functional and logic programming. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93. ACM Press, 1997.

[14] M. Hanus and P. Réty. Demand-driven search in functional logic programs. Research report RR-LIFO-98-08, Univ. Orléans, 1998.

[15] M. Hanus and F. Steiner. Controlling search in declarative programs. In *Principles of Declarative Programming (Proceedings Joint International Sy mposium PLILP/ALP'98)*, pages 374–390. Springer Lecture Notes in Computer Science 1490, 1998.

[16] M. Hanus (ed.). Curry: An integrated functional logic language. Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, 1998.

[17] P. Hudak, S. Peyton Jones, and P. Wadler. Report on the programming language Haskell (Version 1.2). *SIGPLAN Notices*, 27(5), 1992.

[18] J. Hughes. Why functional programming matters. In D.A. Turner, editor, *Research Topcis in Functional Programming*, pages 17–42. Addison Wesley, 1990.

[19] S. Janson and S. Haridi. Kernal Andorra Prolog and its computation model. In *Proceedings of the 7th International Conference*, pages 31–46. MIT Press, 1990.

[20] S. Janson and S. Haridi. Programming paradigms of the Andorra Kernel Language. In *Proceedings 1991 International Logic Programming Symposium*, pages 167–183. MIT Press, 1991.

[21] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.

[22] R. Loogen. Relating the implementation techniques of functional and functional logic languages. *New Generation Computing*, 11:179–215, 1993.

[23] R. Loogen, F. Lopez Fraguas, and M. Rodríguez Artalejo. A demand driven computation strategy for lazy narrowing. In *Proceedings of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pages 184–200. Springer Lecture Notes in Computer Science 714, 1993.

[24] J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic programming with functions and predicates: The language BABEL. *Journal of Logic Programming*, 12:191–223, 1992.

[25] L. Naish. *Negation and Control in Prolog.* Springer Lecture Notes in Computer Science 238, 1987.

[26] J.F. Nilsson. On the compilation of a domain-based Prolog. In *Proceedings IFIP'83*, pages 293–298. Elsevier Science Publishers, 1983.

[27] M. Odersky and P. Wadler. Pizza into Java: translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 146–159, 1997.

[28] S.L. Peyton Jones and P. Wadler. Imperative functional programming. In *Proceedings 20th Symposium on Principles of Programming Languages (POPL'93)*, pages 71–84, 1993.

[29] E. Pontelli and G. Gupta. Implementation mechanisms for dependent and-parallelism. In *Proceedings of the Fourteenth International Conference on Logic Programming (ICLP'97)*, pages 123–137. MIT Press, 1997.

[30] V.A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.

[31] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, pages 324–343. Springer Lecture Notes in Computer Science 1000, 1995.

[32] Sun Microsystems. Java documentation. Available at `http://java.sun.com/docs/`, 1997.

[33] P. Wadler. How to declare an imperative. In *Proceedings of the 1995 International Logic Programming Symposium*, pages 18–32. MIT Press, 1995.

[34] D. Wakeling. A Haskell to Java Virtual Machine code compiler. In *Proceedings 9th International Workshop on Implementation of Functional Languages (IFL'97)*, pages 39–52. Springer Lecture Notes in Computer Science 1467, 1997.

[35] D.H.D. Warren. Higher-order extensions to Prolog: are they needed? In *Machine Intelligence 10*, pages 441–454, 1982.

[36] D.H.D. Warren. An abstract Prolog instruction set. Technical note 309, SRI International, Stanford, 1983.

A Description of the Abstract Machine

This appendix contains a more detailed description of the structure and the commands of our abstract machine.

A.1 Registers and Data Structures of Individual Machines

We recall the registers and data structures as introduced and motivated in the paper. Each machine contains the following registers and data areas:

`AReg`$_0$,`AReg`$_1$,`...`**:** Argument registers for the actual arguments.

`TReg`$_0$,`TReg`$_1$,`...`**:** Auxiliary registers for pattern matching.

`PC`**:** Program counter pointing to the next instruction. `PC` consists of two components `pcFunction` (the function that the machine is executing) and `pcIndex` (the next command in the `code` block of `pcFunction`).

`TermStack`**:** Small stack to construct new terms as needed by the right-hand sides of function rules.

`CallStack`**:** Stack where the state of the machine is stored each time a function is called. Thus, the state can be restored when the machine returns from the function evaluation. An entry in this stack consists of:

- a reference to the function call to be evaluated
- a copy of the relevant argument registers
- a return address
- the index of an argument register where the result should be stored

`key`**:** The key of this machine for variable bindings (an integer value).

`birthTimeStamp`**:** The timestamp when this machine was created (an integer value).

`timeStamp`**:** The current timestamp of this machine (an integer value).

`CompStructure`**:** A reference to the enclosing computation structure.

A.2 Commands of the Abstract Machine

This section describes all commands of the abstract machine and explains their influence on the different registers and data areas. The program counter `pcIndex` is implicitly incremented by one before the command is executed (i.e., it always points to the subsequent instruction if it is not set by this command).

A.2.1 Commands changing the Execution Path

$\boxed{\texttt{CmdJump(}\textit{index}\texttt{)}}$

Local jump in the code block of a function:

```
pcIndex := $index$
```

$\boxed{\texttt{CmdCall(}r\texttt{,}\textit{NumSave}\texttt{,}\textit{ContIndex}\texttt{)}}$

Evaluate the function call referred by register $\texttt{AReg}_r$ and save the first *NumSave* argument registers. After evaluation, jump to *ContIndex*:

```
let $f(a_1, \ldots, a_n)$ be the function call referred by $\texttt{AReg}_r$
if   result table of $f(a_1, \ldots, a_n)$ contains an entry $result$
     for the current machine
then $\texttt{AReg}_r$ := $result$
     pcIndex := $ContIndex$
else pcIndex := $ContIndex$
     create entry on CallStack and save into its components:
          reference to $f(a_1, \ldots, a_n)$
          $\texttt{AReg}_0$,...,$\texttt{AReg}_{NumSave-1}$
          PC
          $r$
     $\texttt{AReg}_0$,...,$\texttt{AReg}_{n-1}$ := $a_1$,...,$a_n$
     pcFunction := $f(a_1, \ldots, a_n)$
     pcIndex := 0
```

$\boxed{\texttt{CmdExecute}}$

Evaluate the function call referred by the top element of the term stack. This command is used for newly created function calls which must be immediately evaluated since they occur at the top-level in the right-hand side of a rule.

```
$term$ := pop(TermStack)
let $f(a_1, \ldots, a_n)$ be the function call referred by $term$
$\texttt{AReg}_0$,...,$\texttt{AReg}_{n-1}$ := $a_1$,...,$a_n$
pcFunction := $f(a_1, \ldots, a_n)$
pcIndex := 0
```

$\boxed{\texttt{CmdReturn}}$

Return from the evaluation of a function call:

```
$result$ := pop(TermStack)
load the components of top(CallStack) into (and pop it):
        reference to $f(a_1, \ldots, a_n)$
        PC
        $\texttt{AReg}_0, \ldots, \texttt{AReg}_m$
        $r$
store $result$ in result table of $f(a_1, \ldots, a_n)$
$\texttt{AReg}_r$ := $result$
```

A.2.2 Operations on the Term Stack

$\boxed{\texttt{CmdPushCons(}Name\texttt{,}NumArgs\texttt{)}}$

Create a constructor term and push it on the term stack

$\boxed{\texttt{CmdPushFunc(}Name\texttt{,}NumArgs\texttt{)}}$

Create a function term and push it on the term stack:

```
$a_{NumArgs}$ := pop(TermStack)
...
$a_1$ := pop(TermStack)
create new object $Name(a_1, \ldots, a_{NumArgs})$ and push reference to this
on the TermStack
```

$\boxed{\texttt{CmdPushVar}}$

Create new variable object and push a reference on the term stack:

```
create new variable object (with an empty binding table)
push reference to this new object on the TermStack
```

$\boxed{\texttt{CmdPushReg(}r\texttt{)}}$

Push argument register r on the term stack:

```
push register $\texttt{AReg}_r$ on the TermStack
```

$\boxed{\texttt{CmdPushPrimInt(}N\texttt{)}}$

Push an integer constant on the term stack (similarly for other primitive types):

push integer N on the `TermStack`

A.2.3 Register Operations

$\boxed{\texttt{CmdCopyReg(}s\texttt{,}d\texttt{)}}$

Copy argument registers:

$\texttt{AReg}_d$ `:=` $\texttt{AReg}_s$

$\boxed{\texttt{CmdNewVar(}r\texttt{)}}$

Create new variable object and store reference in register r:

$\texttt{AReg}_r$ `:=` reference to new variable object

$\boxed{\texttt{CmdLoadReg(}r\texttt{,}t\texttt{)}}$

Copy temporary register t into an argument register r:

$\texttt{AReg}_r$ `:=` $\texttt{TReg}_t$

$\boxed{\texttt{CmdLoadArgs(}r\texttt{)}}$

Load arguments of constructor object into temporary registers:

```
let $c(a_1, \ldots, a_n)$ be the constructor object referred by $\texttt{AReg}_r$
$\texttt{TReg}_1$,...,$\texttt{TReg}_n$ := $a_1$,...,$a_n$
```

A.2.4 Implementing Non-Determinism and Pattern Matching

$\boxed{\texttt{CmdNewChoice(}i\texttt{,}\mathit{NumArgs}\texttt{)}}$

Create a new machine that can follow a different alternative in the computation:

```
Create a copy of the current computation structure
(i.e., the AND-tree to which the current machine belongs)
Set the initial state of the new machine (representing the copy of
this machine) to:
 $\texttt{AReg}_1$,...,$\texttt{AReg}_{NumArgs}$ := $\texttt{AReg}_1$,...,$\texttt{AReg}_{NumArgs}$ of current machine
 TermStack := emptyStack
 CallStack := CallStack of current machine
 pcFunction := pcFunction of current machine
 pcIndex := $i$
```

$\boxed{\texttt{CmdOr(}i\texttt{,}\mathit{NumArgs}\texttt{)}}$

Create a new machine (used for overlapping left-hand sides):

This command tests if there are other active machines in the AND-tree. If not, it works like `CmdNewChoice`. Otherwise, it forces the current machine to suspend.

$\boxed{\texttt{CmdSwitchOnType(}r\texttt{,}\mathit{isFuncIndex}\texttt{,}\mathit{isVarIndex}\texttt{)}}$

Case distinction on the type of the term referred by register r:

```
if $\texttt{AReg}_r$ refers to a variable then
      if binding table of $\texttt{AReg}_r$ contains an entry for the current machine
      then $\texttt{AReg}_r$ := binding of $\texttt{AReg}_r$
if $\texttt{AReg}_r$ refers to a function
then pcIndex := $\mathit{isFuncIndex}$
else if $\texttt{AReg}_r$ refers to a variable then
            if   there are other active machines in the current
                 computation structure
            then suspend this machine;
                 mark this variable so that this machine will be
                 reactivated when the variable will be bound;
                 do not increment the program counter (so the
                 machine will restart executing this command again)
            else
                 pcIndex := $\mathit{isVarIndex}$
```

$\boxed{\texttt{CmdSwitchOnDet(}r\texttt{,}\mathit{isFuncIndex}\texttt{,}\mathit{isVarIndex}\texttt{)}}$

This command is used instead of `CmdSwitchOnType` if the variable case binds the variable deterministically:

```
Execution:
Like CmdSwitchOnType but this command does not suspend
if other active machines are present
```

$\boxed{\texttt{CmdSwitchOnRigid(}r\texttt{,}\mathit{isFuncIndex}\texttt{)}}$

This command is used to implement rigid functions:

```
Execution:
Like CmdSwitchOnType but this command always suspends
if the (dereferenced) $\texttt{AReg}_r$ is a variable object
```

$\boxed{\texttt{CmdSwitchOnCons(}r\texttt{,}\mathit{JumpTable}\texttt{)}}$

Case distinction on constructors:

```
let $\texttt{AReg}_r$ be a reference to constructor object $c(a_1, \ldots, a_n)$
where the constructor $c$ has index $i$
if $\mathit{JumpTable}$[$i$] = -1 then
      the computation of the current machine fails
else
      pcIndex := $\mathit{JumpTable}$[$i$]
```

A.2.5 Binding Variables

$\boxed{\texttt{CmdSimpleBind(}r\texttt{)}}$

Bind variable in register r to top element of term stack:

```
let $var$ be the variable object referred by $\texttt{AReg}_r$
$term$ := pop(TermStack)
add a binding entry for $term$ in binding table of $var$ (with key and
timeStamp of current machine)
activate suspended machines in the current computation structure
that wait on this variable
```

$\boxed{\texttt{CmdDetBind}(r)}$

Bind variable deterministically (only used in combination with `CmdSwitchOnDet`):

> Execution:
> Like `CmdSimpleBind` but since it is only used in combination with `CmdSwitchOnDet`, it performs some optimizations due to the fact that nobody else tries to bind the variable

A.2.6 Custom commands

$\boxed{\texttt{CmdCustom}(mname)}$

This command is used for the optimization explained at the end of Section 3.1 to avoid operations on the term stack. It calls the Java method *mname* (a method of the current function). This method must have the type `Machine`→`void`. During the execution of the method, parts of the AND-tree administration are disabled for this machine. Thus, it is important that no change of the execution path is made inside the method.
The method can use its argument of type `Machine` to access components of the executing machine with the following methods:

`Term getReg(int` r`)`	returns the contents of $\texttt{AReg}_r$
`void returnWith(Term result)`	see `CmdReturn`
`void continueWith(Function func)`	see `CmdExecute`
`Variable allocNewVariable()`	see `CmdPushVar`

Thus, arguments can be directly passed to the machine without using the `TermStack`.