A Concurrent Implementation of Curry in Java

Michael Hanus Ramin Sadre

RWTH Aachen, Informatik II, D-52056 Aachen, Germany

{hanus,ramin}@i2.informatik.rwth-aachen.de

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 sketches a first prototype implementation of Curry in Java. The main emphasis of this 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

1 Introduction

Curry [11, 12] 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 [9] for a survey on functional logic programming).

Curry's operational semantics is based on a single computation model, described in [11], 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 [20] 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 them 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 [10].

In order to provide a portable implementation of Curry, we have developed an interpreter for Curry in Java. Each function defined in a Curry program is translated into a Java class containing instructions of an abstract machine for Curry programs.[1] These instructions implement the pattern matching and control flow of a Curry program. They are executed by a small interpreter implemented in Java.

In this implementation we exploit the concurrent features of Java in two ways. *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 processes for don't know non-deterministic computations which is also known as *stability* from AKL [15]. This scheme implements coroutining features of current Prolog systems [18] as well as features of concurrent constraint languages [22].

In the next section, we review the basic computation model of Curry. Its concurrent implementation in Java is described in Section 3. Section 4 discusses some extensions of the basic computation model, and Section 5 contains our conclusions.

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 [11, 12].

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

[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 concurrently (by Java threads) rather than sequentially as in Prolog. Thus, standard techniques to translate (sequential) logic programs into procedures of imperative languages (e.g., [19]) cannot be applied here.

[2] In the following we use a Prolog-like syntax which is slightly different from the actual Curry syntax.

construct natural numbers,[3] and polymorphic lists (`A` is a type variable ranging over all types) are defined as in Prolog.

A *data term* is a well-formed expression containing variables, constants and data constructors, e.g., `s(s(z))`, `[true|Z]` etc. *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 *equations* (or *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 equation* has the form l `|` $\{c\}$ `=` r, where the condition c is a constraint consisting of a conjunction of equations. A conditional equation 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 equations.

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                        leq(z   ,X   ) = true
add(s(X),Y) = s(add(X,Y))              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:

```
concat([]    ,Ys) = Ys
concat([X|Xs],Ys) = [X|concat(Xs,Ys)]
last(Xs) | {concat(Ys,[X])=Xs} = X
```

If the equation `concat(Ys,[X])=Xs` is solvable, then `X` is the last element of the list `Xs`. □

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(s(z)),s(z))` by applying the rules for addition to this expression:

```
add(s(s(z)),s(z)) → s(add(s(z),s(z))) → s(s(add(z,s(z)))) → s(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 [13].

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

[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.

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"

$$\begin{aligned} \texttt{true} \vee \texttt{X} &= \texttt{true} \\ \texttt{X} \vee \texttt{true} &= \texttt{true} \\ \texttt{false} \vee \texttt{false} &= \texttt{false} \end{aligned}$$

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.

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(a) = c
f(b) = d
```

(where `a`, `b`, `c`, `d` are constants), then the expression `f(X)` with the free variable `X` is evaluated to `c` or `d` by binding `X` to `a` or `b`, respectively. 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. For instance, in the previous example the evaluation of `f(X)` is reflected by the following (non-deterministic) computation step:

$$\texttt{f(X)} \;\rightarrow\; \{\texttt{X=a}\}\texttt{c} \mid \{\texttt{X=b}\}\texttt{d}$$

(Here `|` denotes a disjunction, $\{\texttt{X=a}\}$ denotes a substitution (binding), and $\{\texttt{X=a}\}\texttt{c}$ 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 corresponding bindings (*non-deterministic step*). For inductively sequential programs [1] (these are, roughly speaking, function definitions without overlapping left-hand sides), this strategy, called *needed narrowing* [2], 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.[5]

[4] Since the exact evaluation strategy of Curry is specified using definitional trees [1, 11], Curry also supports more powerful evaluation strategies than current functional languages.

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

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 [21]), we will implement disjunctive computations by independent 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 functions in Curry.

Functional logic languages are often used to solve equations between expressions containing defined functions. For instance, consider the function `add` defined in Example 1 and the equation `add(X,z)=s(z)`. This equation can be solved by evaluating the left-hand side `add(X,z)` to the answer expression {`X=s(z)`}`s(z)` (here we omit the other alternatives in the disjunction). Since the resulting equation is trivial, the equation is valid w.r.t. the computed answer {`X=s(z)`}. In general, an *equation* or *equational constraint* e_1=e_2 is satisfied if both sides e_1 and e_2 are reducible to a same data term. As a consequence, if both sides are undefined (non-terminating), then the equality does not hold.[6] 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 [16]. 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 pure logic programming. On the other hand, it is also known that the guessing of free variables should not be applied to all functions or predicates, since some of them (in particular, those 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-Boolean functions are marked as rigid and predicates are marked as flexible. Thus, pure logic programs behaves as in Prolog, and pure 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 $\wedge$. If c_1 and c_2 are constraints, $c_1 \wedge c_2$ is a constraint which is evaluated by solving c_1 and c_2 concurrently. In a sequential implementation, $c_1 \wedge c_2$ 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 our concurrent implementation, 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 [18]. For instance, if `gen` is a predicate or constraint which instantiates its argument with potential solutions (i.e., `gen` is flexible) and `test` checks whether the argument is a

[6]This notion of equality is also known as *strict equality* [7, 17] and is the only reasonable notion of equality in the presence of non-terminating functions.

correct solution (i.e., `test` is rigid), then a constraint like "`test(X)`$\wedge$`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 [4] or Goffin [5]. For instance, a constraint of the form "`X=f(t1)`$\wedge$`Y=g(t2)`$\wedge$`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 rigid in all arguments).

The advantage of Curry's computation model is the clear separation between sequential and concurrent parts. Sequential computations, which could be considered as the basic units of a program, could be expressed as usual functional (logic) programs, and they are composed to concurrent computation units via concurrent conjunctions of constraints. Since constraints could 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 [5]. Thus, the same abstraction facilities could 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 [15], CCP [22], or Oz [23].

3 Implementation of Curry Programs

In this section we sketch a Java-based implementation of an abstract machine for executing Curry programs. We describe the implementation of data terms, defined functions, and the computational model of Curry (variable bindings, non-determinism, concurrency).

The main motivation for using Java is to provide a portable implementation of Curry and its concurrency features. Due to the use of Java, it is also evident that this implementation cannot compete with other highly optimized implementations of Prolog. On the other hand, the use of Java provides automatic memory management and support for concurrent programming which largely simplifies the implementation task for Curry.

3.1 Representation of Data Terms and Functions

The run-time system of our implementation is completely written in Java [24]. This has influenced some implementation choices, in particular, the run-time representation of constructors and functions. 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 (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.

In order to simplify the implementation task, we use Java's object system for the representation of dynamically created data terms and expressions. Thus, expressions in Curry are represented as Java objects in a straightforward way. Constructors and functions are instances of the classes `Constructor` and `Function`, respectively. Every instance of this class represent an expression of Curry, where the possible arguments of the expression are references from this instance to other objects representing the arguments. Therefore, these classes have the following structure (`Term` is a superclass of both `Constructor` and `Function`):

```
class Constructor extends Term {
  private ConsDesc cd ;  // infos about name and type
  Term [] args ;         // reference to arguments
  ...
}

abstract class Function extends Term {
  ...
  Term [] args ;         // reference to arguments
  static Command code[] ;
}
```

In the following, we will show the principles of our implementation 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`.

Function calls are represented as instances of a subclass of `Function`. Each function object has a component `code` containing the instruction sequence to be executed if this function call must be evaluated (due to the `static` declaration, the code is represented only once for all instances). Thus, the Curry compiler translates each function definition into a definition of a subclass of `Function`. For instance, the function `add` is translated into the class `Function_add` which contains in the component `code` the corresponding instructions for an abstract Curry machine, which will be described below.

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. It searches for the class `Function_add` and loads it into memory (loading of classes during run-time is supported by the JVM).
2. An instance of the class 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`.
3. 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 class `Function_add`.

Now we discuss more details of the code instructions. 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. 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:

- Argument registers that holds the actual arguments of a function call.
- Some auxiliary registers for pattern matching, i.e., to get access to the arguments of constructor terms (see "`T`" in the pseudo code above).
- A result register containing the result of a function call.
- A program counter pointing to the next instruction.
- A small stack for the construction of expressions.
- A call stack to store the actual machine state when a function is called.

Single commands are represented by instances of classes that are derived from the abstract class `Command`:

```
abstract 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 register 0, Argument 2 in register 1
     new CmdSwitchOnType(0,11),       // register 0 is function -> line 11
 // code for case 'isConstructor'
     new CmdSwitchOnCons(0,{2,4}),    // Constructor: z -> line 2, s -> line 4
 // code for rule 1, line 2:
     new CmdPushReg(1),
     new CmdReturn(),                 // return register 1 as result
 // code for rule 2, line 4:
     new CmdLoadArgs(0),
     new CmdLoadReg(0,0),             // get argument of constructor: 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', line 11:
     new CmdCall(0,2),                // execute function call in register 0
     new CmdGetResult(0),
     new CmdJump(0)                   // and try again
```

```
  };
}
```

This is only a brief description of the abstract machine. The current implementation additionally performs some optimizations (e.g., right-hand sides of rules which do not contain control transfer instructions are directly compiled into Java code in order to avoid the stepwise execution of the command sequence).

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 are 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)`}. □

The run-time system of our implementation uses threads to implement different abstract machines. Each machine needs a data structure that stores the bindings initiated by the machine (see [8] for an overview of different implementation methods). Our current implementation is based on *binding arrays*: every variable has a unique number that is interpreted as an index in an array of expressions. This array (the binding array) contains the expressions the variables are bound to. Thus, variables are represented as instances of the class `Variable` containing the index in the binding array:

```
class Variable {
  int index ;
  ...
}
```

The binding array method guarantees constant variable access but results in a non-constant machine creation time: in fact it is linear in the number of used logical variables. In contrast to logic programming, this is not a problem in a functional logic language: free variables do not occur so often in application programs compared to logic programs, since they are not necessary for passing intermediate values due to the use of nested expressions. Moreover, the larger part of Curry programs are purely functional computations which do not need free variables and non-deterministic bindings at all.

In order to extend the code for the `add` example of the previous section to include the possible binding of free variables, we replace the first command (`SwitchOnType`) by the instruction

```
SwitchOnType(0,11,14)    // register 0: function -> line 11, variable -> 14
```

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 `'isVariable'`) at the end of the previous code for `add` (here we show the generated code instead of the creation commands "`new Cmd...`"):

```
// handling of free variable as first argument (this is line 14)
    NewMachine(18),          // start a copy of this machine at line 18
// choice 1:
    PushCons("z",0),
    SimpleBind(0),           // bind register 0 to 'z'
    Jump(2),
// choice 2 (this is line 18)
    PushVar(),               // create new variable X
    PushCons("s",1),
    SimpleBind(0),           // bind register 0 to 's(X)'
    Jump(4)
```

If the first argument of a call to `add` is a free variable, the current computation thread is duplicated. One thread binds the free variable to the constructor `z` and proceeds with the first rule, while the other thread 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 introduced in Section 2, Curry also supports the suspension of function evaluations that are not sufficiently instantiated. As known from logic programming with coroutining [18], 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 also essential to connect *external functions* in a clean way [3]. For instance, the integer numbers can be considered as an infinite set of constants where 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 $\wedge$, a goal like "`X+3=Y` $\wedge$ `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 different threads in Java, where the threads evaluate different constraints. The synchronization of these threads is organized through the binding of common variables. This has the consequence that the run-time system must start new evaluation 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—apart from the binding array that is now shared by all AND-parallel machines—the AND-tree representing the structure of concurrent conjunctions. This tree organizes the AND-parallel evaluation of a goal:

- The leafs represent working machines evaluating a constraint.
- An evaluation of a concurrent conjunction $c_1 \wedge c_2$ 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 thread 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 leafs 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 leafs:

1. *Active leafs* represent running machines.
2. *Suspended leafs* 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 binding array for variables is 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 variable shared by the computation structure. A problem occurs if one thread wants to bind a variable to different values (OR-parallelism). The two extreme solutions to this problem 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 also known as *stability* from AKL [15]:

- Leafs 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 threads and prefers deterministic parts of the computation, similarly to the Andorra computation model [14]. If all leafs in a computation structure are suspended but there are several leafs waiting to bind a variable non-deterministically, then only one of them (e.g., the leaf corresponding to the leftmost constraint) is re-activated.

A small example helps to see how this works. Our program consists of one predicate (i.e., a function that returns a Boolean value):

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

This (flexible) function acts in goals like

```
X+X=Y ∧ X*X=Y ∧ digit(X)=true
```

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 leafs for the constraints `X+X=Y`, `X*X=Y` and `digit(X)=true` 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 leafs.

3. The evaluations of the functions `+` and `*` suspend since all arguments are uninstantiated.

4. Now all consumers are suspended. This permits the machine for `digit(X)` to create bindings for the variable `X`.

5. The leafs 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.

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.

4 Extensions of the Basic Computation Model

Higher-order functions has been shown to be very useful to structure programs and write reusable software [13]. 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 [25] 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 functions, every function call (i.e., instance of a subclass of `Function`) stores the arity of the function *and* the number of arguments currently bound to the function object. If a partial function call (i.e., an object whose number of arguments is less than its arity) is applied to an argument, it simply adds the new argument to the function object and increments the argument counter.[7]

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 [17]: consider a conditional rule "$l \mid \{c\} = r$" (the condition c is a constraint) 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 "$c \Rightarrow r$" is defined by the predefined rule

$$(\{\} \Rightarrow \mathtt{x}) \quad = \quad \mathtt{x} \ .$$

Thus, a guarded expression is evaluated by an attempt to solve the constraint. If this is successful (i.e., leads to the empty constraint $\{\}$), the guarded expression is replaced by the right-hand side r of the conditional rule.

5 Conclusions and Results

We have presented the structure of a first implementation of the multi-paradigm language Curry in Java. 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 want 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 "$\mathtt{generate(X)} \wedge \mathtt{test(X)}$" does not show any difference—neither in result nor in execution time—to the evaluation of "$\mathtt{test(X)} \wedge \mathtt{generate(X)}$". Concerning this, the implementation is comparable with the Andorra model [14].

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. 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 10000 LIPS ("**L**ogical **I**nferences **P**er **S**econd", where a logical inference is here a reduction step with an equation) on a Pentium 100MHz-system. 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. We have also tested the behavior of generating threads to implement the non-deterministic search (OR-parallelism). For this purpose, we executed a highly non-deterministic program (a map coloring problem with 47

[7] Note that the machine has to make a copy of the function object because it may be shared with other OR-parallel computation structures.

solutions) with our system. To compute all solutions for this problem, our implementation created 124 threads during an execution time of 2.2 seconds.

Since Java is relatively new and Java compilers (in particular Just-in-Time-compilers) are 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 and external functions directly implemented in Java.

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 [6]). 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.

Future extensions of this implementation will include methods to restrict and control the non-determinism by a committed choice and the encapsulation of search. 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.

References

[1] S. Antoy. Definitional trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pages 143–157. Springer LNCS 632, 1992.

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

[3] S. Bonnier and J. Maluszynski. Towards a clean amalgamation of logic programs with external procedures. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pages 311–326. MIT Press, 1988.

[4] 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.

[5] M.M.T. Chakravarty, Y. Guo, M. Köhler, and H.C.R. Lock. Goffin - higher-order functions meet concurrent constraints. *Science of Computer Programming (to appear)*, 1997.

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

[7] 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.

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

[9] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[10] M. Hanus. Teaching functional and logic programming with a single computation model. In *Proc. Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP'97)*, pages 335–350. Springer LNCS 1292, 1997.

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

[12] M. Hanus (ed.). Curry: An integrated functional logic language. Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, 1997.

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

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

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

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

[17] 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.

[18] L. Naish. *Negation and Control in Prolog*. Springer LNCS 238, 1987.

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

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

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

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

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

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

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