©Springer-Verlag
In *Proc. of the 3rd International Workshop on Frontiers of Combining Systems, FroCoS 2000*, Nancy.
Springer LNCS 1794, pp. 171–185, 2000

Compiling Multi-Paradigm Declarative Programs into Prolog*

Sergio Antoy[1] Michael Hanus[2]

[1] Department of Computer Science, Portland State University,
P.O. Box 751, Portland, OR 97207, U.S.A., antoy@cs.pdx.edu
[2] Institut für Informatik, Christian-Albrechts-Universität Kiel,
Olshausenstr. 40, D-24098 Kiel, Germany, mh@informatik.uni-kiel.de

Abstract. This paper describes a high-level implementation of the concurrent constraint functional logic language Curry. The implementation, directed by the lazy pattern matching strategy of Curry, is obtained by transforming Curry programs into Prolog programs. Contrary to previous transformations of functional logic programs into Prolog, our implementation includes new mechanisms for both efficiently performing concurrent evaluation steps and sharing common subterms. The practical results show that our implementation is superior to previously proposed similar implementations of functional logic languages in Prolog and is competitive w.r.t. lower-level implementations of Curry in other target languages.

An noteworthy advantage of our implementation is the ability to immediately employ in Curry existing constraint solvers for logic programming. In this way, we obtain with a relatively modest effort the implementation of a declarative language combining lazy evaluation, concurrency and constraint solving for a variety of constraint systems.

1 Introduction

The multi-paradigm language Curry [12, 18] seamlessly combines 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, the language provides both the most important operational principles developed in the area of integrated functional logic languages: "residuation" and "narrowing" (see [10] for a survey on functional logic programming).

Curry's operational semantics (first described in [12]) combines lazy reduction of expressions with a possibly non-deterministic binding of free variables occurring in expressions. To provide the full power of logic programming, (equational) constraints can be used in the conditions of function definitions. Basic constraints can be combined into complex constraint expressions by a concurrent conjunction operator that evaluates

* This research has been partially supported by the DAAD/NSF under grant INT-9981317, the German Research Council (DFG) under grant Ha 2457/1-1, and by a grant from Portland State University.

constraints concurrently. Thus, purely functional programming, purely logic programming, and concurrent (logic) programming are obtained as particular restrictions of this model [12].

In this paper, we propose a high-level implementation of this computation model in Prolog. This approach avoids the complex implementation of an abstract machine (e.g., [16]) and is able to reuse existing constraint solvers available in Prolog systems. In the next section, we review the basic computation model of Curry. The transformation scheme for compiling Curry programs into Prolog programs is presented in Section 3. Section 4 contains the results of our implementation. Section 5 discusses related work and contains our conclusions.

2 The Computation Model of Curry

This section outlines the computation model of Curry. A formal definition can be found in [12, 18].

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 by data type declarations such as:[1]

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

True and **False** are the Boolean constants. **[]** (empty list) and **:** (non-empty list) are the constructors for polymorphic lists (**a** is a type variable ranging over all types and the type **List a** is usually written as **[a]** for conformity with Haskell). A *data term* is a well-formed expression containing variables, constants and data constructors, e.g., **True:[]** or **[x,y]** (the latter stands for **x:(y:[])**).

Functions are operations on data terms whose meaning is specified by (*conditional*) *rules* of the general form "l **|** c **=** r **where** vs **free**". l has the form $f\,t_1 \ldots t_n$, where f is a function, $t_1, \ldots, t_n$ are data terms and each variable occurs only once. The *condition* c is a constraint. r is a well-formed *expression* that may also contain function calls. vs is the list of *free variables* that occur in c and r, but not in l. The condition and the **where** part can be omitted if c and vs are empty, respectively. A *constraint* is any expression of the built-in type **Constraint**. Primitive constraints are equations of the form e_1 **=:=** e_2. A conditional rule is applied only if its condition is satisfiable. A *Curry program* is a set of data type declarations and rules.

Example 1. Together with the above data type declarations, the following rules define operations to concatenate lists and to find 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 "**conc ys [x] =:= xs**" is solvable, then **x** is the last element of list **xs**. □

[1] Curry has a Haskell-like syntax [25], 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").

Functional programming: In functional languages, the interest is 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 replacing instances of rules' left sides with corresponding instances of right sides. For instance, we compute the value of "**`conc [1] [2]`**" by repeatedly applying the rules for concatenation to this expression:

```
conc [1] [2]  →  1:(conc [] [2])  →  [1,2]
```

Curry is based on a lazy (outermost) strategy, i.e., the selected function call in each reduction step is outermost among all reducible function calls. This strategy supports computations with infinite data structures and a modular programming style with separation of control aspects. Moreover, it yields optimal computations [5] and a demand-driven search method [15] for the logic part of a program which will be discussed next.

Logic programming: In logic languages, an expression (or constraint) may contain free variables. A logic programming system should compute solutions, i.e., find values for these variables such that the expression (or constraint) is reducible to some value (or satisfiable). Fortunately, this requires only a minor extension of the lazy reduction strategy. The extension deals with non-ground expressions and variable instantiation: if the value of a free variable is demanded by the left-hand sides of some program rules in order to continue the computation (i.e., no program rule is applicable if the variable remains unbound), the variable is bound to all the demanded values. For each value, a separate computation is performed. For instance, if the function **`f`** is defined by the rules

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

(the integer numbers are considered as an infinite set of constants), then the expression "**`f x`**", with **`x`** a free variable, is evaluated to **`2`** by binding **`x`** to **`0`**, or it is evaluated to **`3`** by binding **`x`** to **`1`**. Thus, a single computation step 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 [3] (these are, roughly speaking, function definitions with one demanded argument), this strategy is called *needed narrowing* [5]. Needed narrowing computes the shortest successful derivations (if common subterms are shared) and minimal sets of solutions. Moreover, it is fully deterministic for expressions that do not contain free variables.

Constraints: In functional logic programs, it is necessary to solve equations between expressions containing defined functions (see Example 1). 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 value (data term). As a consequence, if both sides are undefined (non-terminating), then the equality does not hold.[2] 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 [21]. Thus, an equational constraint e_1**`=:=`**e_2 without occurrences of defined functions has the same meaning (unification) as in Prolog. Curry's basic kernel only provides

[2] This notion of equality, known as *strict equality* [9, 22], is the only reasonable notion of equality in the presence of non-terminating functions.

equational constraints. Constraint solvers for other constraint structures can be conceptually integrated without difficulties. The practical realization of this integration is one of the goals of this work.

Concurrent computations: To support flexible computation rules and avoid an uncontrolled instantiation of free argument variables, Curry gives the option to *suspend a function call* if a demanded argument is not instantiated. Such functions are called *rigid* in contrast to *flexible* functions—those that instantiate their arguments when the instantiation is necessary to continue the evaluation of a call. As a default easy to change, Curry's constraints (i.e., functions with result type **`Constraint`**) are flexible whereas non-constraint functions are rigid. Thus, purely logic programs (where predicates correspond to constraints) behave as in Prolog, and purely functional programs are executed as in lazy functional languages, e.g., Haskell.

To continue a computation in the presence of suspended function calls, constraints are combined with the *concurrent conjunction* operator **`&`**. The constraint c_1 **`&`** c_2 is evaluated by solving c_1 and c_2 concurrently.

A design principle of Curry is the clear separation of sequential and concurrent activities. Sequential computations, which form the basic units of a program, are expressed as usual functional (logic) programs and are composed into concurrent computation units via concurrent conjunctions of constraints. This separation supports the use of efficient and optimal evaluation strategies for the sequential parts. Similar techniques for the concurrent parts are not available. This is in contrast to other more fine-grained concurrent computation models like AKL [19], CCP [27], or Oz [28].

Monadic I/O: To support real applications, the monadic I/O concept of Haskell [29] has been adapted to Curry to perform I/O in a declarative manner. In the monadic approach to I/O, an interactive program is considered as a function computing a sequence of actions which are applied to the outside world. An *action* has type "**`IO`** α", which means that it returns a result of type α whenever it is applied to a particular state of the world. For instance, **`getChar`**, of type "**`IO Char`**", is an action whose execution, i.e., application to a world, reads a character from the standard input. Actions can be composed only sequentially in a program and their composition is executed whenever the main program is executed. For instance, the action **`getChar`** can be composed with the action **`putChar`** (which has type **`Char -> IO ()`** and writes a character to the terminal) by the sequential composition operator **`»=`** (which has type **`IO`** α **`->`** **`(`**α **`->`** **`IO`** β**`)`** **`->`** **`IO`** β**`)`**. Thus, "**`getChar »= putChar`**" is a composed action which prints the next character of the input stream on the screen. The second composition operator, **`»`**, is like **`»=`**, but ignores the result of the first action. Furthermore, **`done`** is the "empty" action which does nothing (see [29] for more details). For instance, a function which takes a string (list of characters) and produces an action that prints the string to the terminal followed by a new line is defined as follows:

```
putStrLn []     = putChar '\n'
putStrLn (c:cs) = putChar c » putStrLn cs
```

In the next section, we will describe a transformation scheme to implement this computation model in Prolog.

3 A Transformation Scheme for Curry Programs

As mentioned above, the evaluation of nested expressions is based on a lazy strategy. The exact strategy is specified via definitional trees [3], a data structure for the efficient selection of the outermost reducible expressions. Direct transformations of definitional trees into Prolog (without an implementation of concurrency features) have been proposed in [2, 4, 11, 21]. Definitional trees deal with arbitrarily large patterns and use the notion of "position" (i.e., a sequence of positive integers) to specify the subterm where the next evaluation step must be performed. We avoid this complication and obtain a simpler transformation by first compiling definitional trees into case expressions as described, e.g., in [14]. Thus, each function is defined by exactly one rule in which the right-hand side contains case expressions to specify the pattern matching of actual arguments. For instance, the function **`conc`** in Example 1 is transformed into:

```
conc xs ys = case xs of []     -> ys
                        (z:zs) -> z : conc zs ys
```

A case expression is evaluated by reducing its first argument to a *head normal form*, i.e., a term which has no defined function symbol at the top, and matching this reduced term with one of the patterns of the case expression. Case expressions are used for both rigid and flexible functions. Operationally, **`case`** expressions are used for rigid functions only, whereas **`flexcase`** expressions are used for flexible functions. The difference is that a **`case`** expression suspends if the head normal form is a free variable, whereas a **`flexcase`** expression (don't know non-deterministically) instantiates the variable to the different constructors in the subsequent patterns.

To implement functions with overlapping left-hand sides (where there is no single argument on which a case distinction can be made), there is also a *disjunctive expression* "e_1 **`or`** e_2" meaning that both alternatives are don't know non-deterministically evaluated.[3] For instance, the function

```
0 * x = 0
x * 0 = 0
```

is transformed into the single rule

```
x * y = or (flexcase x of 0 -> 0)
           (flexcase y of 0 -> 0)
```

under the assumption that "**`*`**" is a flexible operation.

Transformation schemes for programs where all the functions are flexible have been proposed in [2, 4, 11, 21]. These proposals are easily adaptable to our representation using **`case`** and **`or`** expressions. The challenge of the implementation of Curry is the development of a transformation scheme that provides both the suspension of function calls and the concurrent evaluation of constraints (which will be discussed later).

3.1 Implementing Concurrent Evaluations

Most of the current Prolog systems support coroutining and the delaying of literals [23] if some arguments are not sufficiently instantiated. One could use these features to pro-

[3] In the implementation described in this paper, don't know non-determinism is implemented via backtracking as in Prolog.

vide the suspension, when required, of calls to rigid functions. However, in conditional rules it is not sufficient to delay the literals corresponding to suspended function calls. One has to wait until the condition has been completely proved to avoid introducing unnecessary computations or infinite loops. The following example helps in understanding this problem.

Example 2. Consider the function definitions

```
f x y | g x =:= y  = h y
g [] = []
h [] = []
h (z:zs) = h zs
```

where `g` is rigid and `h` is flexible. To evaluate the expression "`f x y`" (where `x` and `y` are free variables), the condition "`g x =:= y`" must be proved. Since `g` is rigid, this evaluation suspends and the right-hand side is not evaluated. However, if we only delay the evaluation of the condition and proceed with the right-hand side, we run into an infinite loop by applying the last rule forever. This loop is avoided if `x` is eventually instantiated by another thread of the entire computation. □

To explain how we solve this problem we distinguish between sequential and concurrent computations. A sequential computation is a sequence of calls to predicates. When a call is activated, it may return for two reasons: either the call's computation has completed or the call's computation has been suspended or delayed. In a sequential computation, we want to execute a call only if the previous call has completed. Thus, we add an input argument and an output argument to each predicate. Each argument is a variable that is either uninstantiated or bound to a constant—by convention the symbol `eval` that stand for "fully evaluated". We use these arguments as follows. In a sequential computation, the call to a predicate is executed if and only if its input argument is instantiated to `eval`. Likewise, a computation has completed if and only if its output argument is instantiated to `eval`. As one would expect, we chain the output argument of a call to the input argument of the next call to ensure the sequentiality of a computation.

The activation or delay of a call is easily and efficiently controlled by `block` declarations.[4] For instance, the `block` declaration "`:- block f(?,?,?,-,?)`" specifies that a call to `f` is delayed if the fourth argument is a free variable. According to the scheme just described, we obtain the following clauses for the rules defining the functions `f` and `g` above:[5]

```
:- block f(?,?,?,-,?).
f(X,Y,Result,Ein,Eout) :- eq(g(X),Y,Ein,E1), h(Y,Result,E1,Eout).
:- block g(?,?,-,?).
g(X,Result,Ein,Eout) :- hnf(X,HX,Ein,E1), g_1(HX,Result,E1,Eout).
```

[4] An alternative to `block` is `freeze` which leads to a simpler transformation scheme. However, our experiments indicate that `freeze` is a more expensive operation (at least in Sicstus-Prolog Version 3#5). Using `freeze`, the resulting Prolog programs were approximately six times slower than using the scheme presented in this paper.

[5] As usual in the transformation of functions into predicates, we transform n-ary functions into $n+1$-ary predicates where the additional argument contains the result of the function call.

```
:- block g_1(-,?,?,?), g_1(?,?,-,?).
g_1([],[],E,E).
```

The predicate **hnf** computes the head normal form of its first argument. If argument **HX** of **g** is bound to the head normal form of **X**, we can match this head normal form against the empty list with the rule for **g_1**.

We use **block** declarations to control the rigidity or flexibility of functions, as well. Since **g** is a rigid function, we add the block declaration **g_1(-,?,?,?)** to avoid the instantiation of free variables. A computation is initiated by setting argument **Ein** to a constant, i.e., expression **(f x y)** is evaluated by goal **f(X,Y,Result,eval,Eout)**. If **Eout** is bound to **eval**, the computation has completed and **Result** contains the computed result (head normal form).

Based on this scheme, the concurrent conjunction operator **&** is straightforwardly implemented by the following clauses (the constant **success** denotes the result of a successful constraint evaluation):

```
&(A,B,success,Ein,Eout) :- hnf(A,HA,Ein,E1), hnf(B,HB,Ein,E2),
                           waitconj(HA,HB,E1,E2,Eout).

?- block waitconj(?,?,-,?,?), waitconj(?,?,?,-,?).
waitconj(success,success,_,E,E).
```

As one can see, predicate **waitconj** waits for the solution of both constraints.

The elements of our approach that most contribute to this simple transformation of Curry programs into Prolog programs are the implementation of concurrency and the use of both **case** and **or** expressions. Each function is transformed into a corresponding predicate to compute the head normal form of a call to this function. As shown above, this predicate contains additional arguments for storing the head normal form and controlling the suspension of function calls. Case expressions are implemented by evaluating the case argument to head normal form. We use an auxiliary predicate to match the different cases. The difference between **flexcase** and **case** is only in the block declaration for the case argument. "**or**" expressions are implemented by alternative clauses and all other expressions are implemented by calls to predicate **hnf**, which computes the head normal form of its first argument. Thus, function **conc** in Example 1 is transformed into the following Prolog clauses:

```
:- block conc(?,?,?,-,?).
conc(A,B,R,Ein,Eout) :- hnf(A,HA,Ein,E1), conc_1(HA,B,R,E1,Eout).

:- block conc_1(-,?,?,?,?), conc_1(?,?,?,-,?).
conc_1([]    ,Ys,R,Ein,Eout) :- hnf(Ys             ,R,Ein,Eout).
conc_1([Z|Zs],Ys,R,Ein,Eout) :- hnf([Z|conc(Zs,Ys)],R,Ein,Eout).
```

Should **conc** be a flexible function, the block declaration **conc_1(-,?,?,?,?)** would be omitted, but the rest of the code would be unchanged. The definition of **hnf** is basically a case distinction on the different top-level symbols that can occur in an expression and a call to the corresponding function if there is a defined function at the top (compare [11, 21]).

Although this code is quite efficient due to the first argument indexing of Prolog implementations, it can be optimized by partially evaluating the calls to **hnf**, as discussed in [4, 11]. Further optimizations could be done if it is known at compile time that the evaluation of expressions will not cause any suspension (e.g., when all arguments are ground at run time). In this case, the additional two arguments in each predicate and the block declarations can be omitted and we obtain the same scheme as proposed in [11]. This requires static analysis techniques for Curry which is an interesting topic for further research.

3.2 Implementing Sharing

Every serious implementation of a lazy language must implement the sharing of common subterms. For instance, consider the rule

```
double x = x + x
```

and the expression "`double (1+2)`". If the two occurrences of the argument `x` in the rule's right-hand side are not shared, the expression `1+2` is evaluated twice. Thus, sharing the different occurrences of a same variable avoids unnecessary computations and is the prerequisite for optimal evaluation strategies [5]. In low level implementations, sharing is usually obtained by graph structures and destructive assignment of nodes [26]. Since a destructive assignment is not available in Prolog, we resort to Prolog's sharing of logic variables. This idea has been applied, e.g., in [8, 11, 20] where the predicates implementing functions are extended by a free variable that, after the evaluation of the function call, is instantiated to the computed head normal form. Although this avoids the multiple evaluation of expressions, it introduce a considerable overhead when no common subterms occur at run time—in some cases more than 50%, as reported in [11]. Therefore, we have developed a new technique that causes no overhead in all practical experiments we performed. As seen in the example above, sharing is only necessary if terms are duplicated by a variable having multiple occurrences in a condition and/or right-hand side. Thus, we share these occurrences by a special `share` structure containing the computed result of this variable. For instance, the rule of `double` is translated into

```
double(X,R,E0,E1) :- hnf(share(X,EX,RX)+share(X,EX,RX),R,E0,E1).
```

In this way, each occurrence of a left-hand side variable `X` with multiple occurrences in the right-hand side is replaced by `share(X,EX,RX)`, where `RX` contains the result computed by evaluating `X`. `EX` is bound to some constant if `X` has been evaluated. `EX` is necessary because expressions can also evaluate to variables in a functional logic language. Then, the definition of `hnf` is extended by the rule:

```
hnf(share(X,EX,RX),RX,E0,E1) :- !,
  (nonvar(EX) -> E1=E0
              ; hnf(X,HX,E0,E1), EX=eval, propagate-
Share(HX,RX)).
```

where `propagateShare(HX,RX)` puts `share` structures into the arguments of `HX` (yielding `RX`) if `HX` is bound to a structure and the arguments are not already shared.

This implementation scheme has the advantage that the Prolog code for rules without multiple variable occurrences remains unchanged and consequently avoids the overhead for such rules (in contrast to [8, 11, 20]). The following table shows the speedup (i.e., the ratio of runtime without sharing over runtime with sharing), the number of reduction steps without (RS1) and with sharing (RS2), and the number of shared variables (SV) in the right-hand side of rules of programs we benchmarked. It is worth to notice that the speedup for the first two goals reported in [11], which uses a different technique, is 0.64 (i.e., a slowdown) and 3.12. These values show the superiority of our technique.

Example:	Speedup	RS1	RS2	# SV
10000≤10000+10000 =:= True	1.0	20002	20002	0
double(double(one 100000)) =:= x	4.03	400015	100009	1
take 25 fibs	6650.0	196846	177	3
take 50 primes	15.8	298070	9867	2
quicksort (quicksort [...])	8.75	61834	3202	2
mergesort [...]	91.5	303679	1057	14

Program analysis techniques are more promising with our scheme than with [8, 11, 20]. For instance, no **share** structures must be introduced for argument variables that definitely do not contain function calls at run time, e.g., arguments that are always uninstantiated or bound to constructor terms.

3.3 Constraints

Equational constraints, denoted e_1**=:=**e_2, are solved by lazily evaluating each side to unifiable data terms. In our translation, we adopt the implementation of this mechanism in Prolog presented in [21]. Basically, equational constraints are solved by a predicate, **eq**, which computes the head normal form of its arguments and performs a variable binding if one of the arguments is a variable.

```
:- block eq(?,?,-,?).
eq(A,B,Ein,Eout) :- hnf(A,HA,Ein,E1), hnf(B,HB,E1,E2),
                    eq_hnf(HA,HB,E2,Eout).

:- block eq_hnf(?,?,-,?).
eq_hnf(A,B,Ein,Eout) :- var(A), !, bind(A,B,Ein,Eout).
eq_hnf(A,B,Ein,Eout) :- var(B), !, bind(B,A,Ein,Eout).
eq_hnf($c$(X$_1$,...,X$_n$),$c$(Y$_1$,...,Y$_n$),Ein,Eout) :- !,
   hnf((X$_1$=:=Y$_1$)&...&(X$_n$=:=Y$_n$),_,Ein,Eout).  % $\forall n$-ary constr. $c$

bind(X,Y,E,E) :- var(Y), !, X=Y.
bind(X,$c$(Y$_1$,...,Y$_n$),Ein,Eout) :- !,  % $\forall n$-ary constructors $c$
   occurs_not(X,Y$_1$),..., occurs_not(X,Y$_n$), X=$c$(X$_1$,...,X$_n$),
   hnf(Y$_1$,HY$_1$,Ein,E$_1$), bind(X$_1$,HY$_1$,E$_1$,E$_2$),
   ...
   hnf(Y$_n$,HY$_n$,E$_{2n-2}$,E$_{2n-1}$), bind(X$_n$,HY$_n$,E$_{2n-1}$,Eout).
```

Due to the lazy semantics of the language, the binding is performed incrementally. We use an auxiliary predicate, **bind**, which performs an occur check followed by an incremental binding of the goal variable and the binding of the arguments.

Similarly, the evaluation of an expression e to its normal form, which is the intended meaning of e, is implemented by a predicate, **nf**, that repeatedly evaluates all e's subexpressions to head normal form.

Apart from the additional arguments for controlling suspensions, this scheme is identical to the scheme proposed in [21]. Unfortunately, this scheme generally causes a significant overhead when one side of the equation is a variable and the other side evaluates to a large data term. In this case, the incremental instantiation of the variable is unnecessary and causes the overhead, since it creates a new data structure and performs an occur check. We avoid this overhead by evaluating to normal form, if possible, the term to which the variable must be bound. To this aim, we replace **bind** with **bind_trynf** in the clauses of **eq_hnf** together with the following new clause:

```
bind_trynf(X,T,Ein,Eout) :- nf(T,NT,Ein,E1),
        (nonvar(E1) -> occurs_not(X,NT), X=NT, Eout=E1
                     ; bind(X,T,Ein,Eout)).
```

If the evaluation to normal form does not suspend, the variable **X** is bound to the normal form by **X=NT**, otherwise the usual predicate for incremental binding is called. Although this new scheme might cause an overhead due to potential re-evaluations, this situation did not occur in all our experiments. In some practical benchmarks, we have measured a speedup up to a factor of 2.

The compilation of Curry programs into Prolog greatly simplifies the integration of constraint solvers for other constraint structures, if the underlying Prolog system offers solvers for these structures. For instance, Sicstus-Prolog includes a solver for an arithmetic constraint over reals, which is denoted by enclosing the constraint between curly brackets. E.g., goal **{3.5=1.7+X}** binds **X** to **1.8**. We make these constraints available in Curry by translating them into the corresponding constraints of Sicstus-Prolog. For instance, the inequational constraint e_1**<**e_2 is translated as follows. First, e_1 and e_2, which might contain user-defined functions or might be variables, are evaluated to their (head) normal forms, say e'_1 and e'_2. Then, the goal **{**e'_1**<**e'_2**}** is called. With this technique, all constraint solvers available in Sicstus-Prolog become available in Curry.

3.4 Further Features

Curry supports standard higher-order constructs such as lambda abstractions and partial applications. In Prolog, the higher-order features of Curry are implemented according to Warren's original proposal [30] to translate higher-order constructs into first-order logic programming. A lambda abstraction is eliminated by transforming it into a top-level definition of a new function. Consequently, the fundamental higher-order construct is a binary function, **apply**, which applies its first argument, a function, to its second argument, the function's intended argument. For each n-ary function or constructor **f**, we introduce $n-1$ constructors with the same name. This enables us to implement the application function with the following Prolog clauses:

```
apply(f(X1,...,Xk),X,f(X1,...,Xk,X),E,E).     % 0 ≤ k < n−1
apply(f(X1,...,Xn−1),X,H,E0,E) :- hnf(f(X1,...,Xn−1,X),H,E0,E).
```

Note that predicate **apply** should be called only for partial applications or applications where it is known at compile time that the first argument is not a defined function or a constructor. In other words, all first-order calls are directly translated without using `apply` as shown in the previous sections. This implementation of `apply` has the advantage that the unique matching clause is found in constant time due to the first argument indexing of Prolog systems. Although the number of `apply` clauses could be high for large applications, and there are alternative schemes that avoid this problem (e.g., [24]), we have found that this scheme causes no problems for programs with several hundred functions.

Monadic I/O is easily implemented by introducing a special constructor (denoted by “`$io`”) to hold the result of an I/O action. For instance, `getChar` is implemented as a procedure which reads a character, c, from standard input and returns the term “`$io` c” whenever it is evaluated. With this approach, both sequential composition operators `»=` and `»` for actions are defined by:

```
($io x) »= fa = fa x
($io _) »  b  = b
```

Thus, the first action is evaluated to head normal form before the second action is applied. This simple implementation has, however, a pitfall. The result of an I/O action should not be shared, otherwise I/O actions will not be executed as intended. For instance, the expressions “`putChar 'X' » putChar 'X'`” and “`let a = putChar 'X' in a » a`” are equivalent but would produce different results with sharing. Luckily, the intended behavior can be obtained by a slight change of the definition of `hnf` so that terms headed by `$io` are not shared.

The primitives of Curry to encapsulate search and define new search strategies [17] cannot be directly implemented in Prolog due to its fixed backtracking strategy. However, one can implement some standard depth-first search strategies of Curry via Prolog’s `findall` and `bagof` primitives.

4 Experimental Results

We have developed a compiler from Curry programs into Prolog programs (Sicstus-Prolog Version 3#5) based on the principles described in this paper. The practical results are quite encouraging. For instance, the execution of the classic “naive reverse” benchmark is executed at the speed of approximately 660,000 rule applications per second on a Linux-PC (Pentium II, 400 Mhz) with Sicstus-3 (without native code). Note that Curry’s execution with a lazy strategy is costlier than Prolog’s execution. Although the development of the compiler is relatively simple, due to the transformation schemes discussed in the paper, our implementation is competitive w.r.t. other high-level and low-level implementations of Curry and similar functional logic languages. We have compared our implementation to a few other implementations of declarative multi-paradigm languages available to us. The following table shows the results of benchmarks for various features of the language.

Program	Prolog	Toy	Java-1	Java-2	UPV-Curry
rev180	50	110	1550	450	43300
twice120	30	60	760	190	40100
qqsort20	20	20	230	45	72000
primes50	80	90	810	190	>2000000
lastrev120	70	160	2300	820	59700
horse	5	10	50	15	200
account	10	n.a.	450	670	2050
chords	220	n.a.	4670	1490	n.a.
Average speedup:		1.77	23.39	13.55	1150.1

All benchmarks are executed on a Sun Ultra-2. The execution times are measured in milliseconds. The column “Prolog” contains the results of the implementation presented in this paper. “Toy” [7] is an implementation of a narrowing-based functional logic language (without concurrency) which, like ours, compiles into Prolog. This implementation is based on the ideas described in [21]. “Java-1” is the compiler from Curry into Java described in [16]. It uses JDK 1.1.3 to execute the compiled programs. “Java-2” differs from the former by using JDK 1.2. This system contains a Just-in-Time compiler. Finally, UPV-Curry [1] is an implementation of Curry based on an interpreter written in Prolog that employs an incremental narrowing algorithm.

Most of the programs, which are small, test various features of Curry. “rev180” reverses a list of 180 elements with the naive reverse function. “twice120” executes the call “`twice (rev` l`)`”, where `twice` is defined by “`twice xs = conc xs xs`” and l is a list of 120 elements. “qqsort20” calls quicksort (defined with higher-order functions) twice on a list of 20 elements. “primes50” computes the infinite list of prime numbers and extracts the first 50 elements. “lastrev120” computes the last element `x` of a list by solving the equation “`conc xs [x] =:= rev [...]`”. “horse” is a simple puzzle that needs some search. “account” is a simulation of a bank account that uses the concurrency features of Curry. “chords”, the largest of our benchmarks, is a musical application [15] that uses encapsulated search, laziness, and monadic I/O.

The comparison with Toy shows that our implementation of the concurrency features does not cause a significant overhead compared to a pure-narrowing-based language. Furthermore, the “account” example, which heavily uses concurrent threads, demonstrates that our implementation is competitive with an implementation based on Java threads. Although the table indicates that our implementation is superior to other available systems, implementations compiling to C or machine languages may be more efficient. However, the development effort of these lower level implementations is much higher.

5 Related Work and Conclusions

The idea of implementing functional logic programs by transforming them into logic programs is not new. An evaluation of different implementations is presented in [11], where it is demonstrated that functional logic programs based on needed narrowing are superior to other narrowing-based approaches. There are several proposals of compilation of needed narrowing into Prolog [4, 11, 21]. All these approaches lack concurrent evaluations. Moreover, the implementation of sharing, similar in all these approaches, is less efficient than in our proposal, as can be verified in the comparison table (see columns "Prolog" and "Toy").

Naish [24] has proposed NUE-Prolog, an integration of functions into Prolog programs obtained by transforming function definitions into Prolog clauses with additional "**when**" declarations. **when** declarations, which are similar in scope to the **block** declarations that we propose, suspend the function calls until the arguments are sufficiently instantiated. The effect of this suspension is that all functions are rigid—flexible functions are not supported. Functions intended to be flexible must be encoded as predicates by flattening. This approach has the drawback that optimal evaluation strategies [5] cannot be employed for the logic programming part of a program. Strict and lazy functions can be freely mixed, which makes the meaning of programs harder to understand (e.g., the meaning of equality in the presence of infinite data structures). NUE-Prolog uses a form of concurrency for suspending function calls, as we do. But it is more restrictive in that there is no possibility to wait for the complete evaluation of an expression. This leads to the undesired behavior discussed in Example 2.

Apart from the efficiency and simplicity of our transformation scheme of Curry into Prolog programs, the use of Prolog as a target language has further advantages. A high-level implementation more easily accomodates the inclusion of additional features. For instance, the implementation of a standard program tracer w.r.t. Byrd's box model [6] requires only the addition of four clauses to each program and two predicate calls for each implemented function. The most important advantage is the reuse of existing constraint solvers available in Prolog, as shown in Section 3.3. Thus, with a limited effort, we obtain a usable implementation of a declarative language that combines constraint solving over various constraint domains, concurrent evaluation and search facilities from logic programming with higher-order functions and laziness from functional programming. The combination of laziness and search is attractive because it offers a modular implementation of demand-driven search strategies, as shown in [15].

Since the compilation time of our implementation is reasonable,[6] this Prolog-based implementation supports our current main development system for Curry programs.[7] This system has been used to develop large distributed applications with sophisticated graphical user interfaces and Web-based information servers that run for weeks without interruption (see [13] for more details). By taking advantage of both the features of our system and already developed code, we can make available on the Internet constraint programming applications in minutes.

References

1. M. Alpuente, S. Escobar, and S. Lucas. UPV-Curry: an Incremental Curry Interpreter. In *Proc. of 26th Seminar on Current Trends in Theory and Practice of Informatics (SOFSEM'99)*, pp. 327–335. Springer LNCS 1725, 1999.

[6] Since the transformation of Curry programs into Prolog is fast, the overall compilation time mainly depends on the time it takes to compile the generated Prolog code. In our tests, it takes only a few seconds even for programs with approximately 2,000 lines of source code.

[7] **http://www-i2.informatik.rwth-aachen.de/~hanus/pacs/**

2. S. Antoy. Non-Determinism and Lazy Evaluation in Logic Programming. In *Proc. Int. Workshop on Logic Program Synthesis and Transformation (LOPSTR'91)*, pp. 318–331. Springer Workshops in Computing, 1991.
3. S. Antoy. Definitional Trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 143–157. Springer LNCS 632, 1992.
4. S. Antoy. Needed Narrowing in Prolog. Technical Report 96-2, Portland State University, 1996.
5. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. Journal of the ACM (to appear). Previous version in *Proc. 21st ACM Symposium on Principles of Programming Languages*, pp. 268–279, 1994.
6. L. Byrd. Understanding the Control Flow of Prolog Programs. In *Proc. of the Workshop on Logic Programming*, Debrecen, 1980.
7. R. Caballero-Roldán, J. Sánchez-Hernández, and F.J. López-Fraguas. User's Manual for TOY. Technical Report SIP 97/57, Universidad Complutense de Madrid, 1997.
8. P.H. Cheong and L. Fribourg. Implementation of Narrowing: The Prolog-Based Approach. In K.R. Apt, J.W. de Bakker, and J.J.M.M. Rutten, editors, *Logic programming languages: constraints, functions, and objects*, pp. 1–20. MIT Press, 1993.
9. E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel LEAF: A Logic plus Functional Language. *Journal of Computer and System Sciences*, Vol. 42, No. 2, pp. 139–185, 1991.
10. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, Vol. 19&20, pp. 583–628, 1994.
11. M. Hanus. Efficient Translation of Lazy Functional Logic Programs into Prolog. In *Proc. Fifth International Workshop on Logic Program Synthesis and Transformation*, pp. 252–266. Springer LNCS 1048, 1995.
12. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages*, pp. 80–93, 1997.
13. M. Hanus. Distributed Programming in a Multi-Paradigm Declarative Language. In *Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, pp. 376–395. Springer LNCS 1702, 1999.
14. M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, Vol. 9, No. 1, pp. 33–75, 1999.
15. M. Hanus and P. Réty. Demand-driven Search in Functional Logic Programs. Research Report RR-LIFO-98-08, Univ. Orléans, 1998.
16. M. Hanus and R. Sadre. An Abstract Machine for Curry and its Concurrent Implementation in Java. *Journal of Functional and Logic Programming*, Vol. 1999, No. 6, 1999.
17. M. Hanus and F. Steiner. Controlling Search in Declarative Programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, pp. 374–390. Springer LNCS 1490, 1998.
18. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.6). Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, 1999.
19. S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *Proc. 1991 Int. Logic Programming Symposium*, pp. 167–183. MIT Press, 1991.
20. J.A. Jiménez-Martin, J. Marino-Carballo, and J.J. Moreno-Navarro. Efficient Compilation of Lazy Narrowing into Prolog. In *Proc. Int. Workshop on Logic Program Synthesis and Transformation (LOPSTR'92)*, pp. 253–270. Springer Workshops in Computing Series, 1992.
21. 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*, pp. 184–200. Springer LNCS 714, 1993.

22. J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The Language BABEL. *Journal of Logic Programming*, Vol. 12, pp. 191–223, 1992.
23. L. Naish. *Negation and Control in Prolog*. Springer LNCS 238, 1987.
24. L. Naish. Adding equations to NU-Prolog. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pp. 15–26. Springer LNCS 528, 1991.
25. J. Peterson et al. Haskell: A Non-strict, Purely Functional Language (Version 1.4). Technical Report, Yale University, 1997.
26. S.L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.
27. V.A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.
28. G. Smolka. The Oz Programming Model. In *Computer Science Today: Recent Trends and Developments*, pp. 324–343. Springer LNCS 1000, 1995.
29. P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, Vol. 29, No. 3, pp. 240–263, 1997.
30. D.H.D. Warren. Higher-order extensions to PROLOG: are they needed? In *Machine Intelligence 10*, pp. 441–454, 1982.