In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*,
Portland (USA), December 1995, pp. 95–107

Curry: A Truly Functional Logic Language

Michael Hanus Herbert Kuchen Juan José Moreno-Navarro
RWTH Aachen* Universidad Politécnica Madrid†

Abstract

Functional and logic programming are the most important declarative programming paradigms, and interest in combining them has grown over the last decade. However, integrated functional logic languages are currently not widely used. This is due to the fact that the operational principles are not well understood and many different evaluation strategies have been proposed which resulted in many different functional logic languages. To overcome this situation, we propose the functional logic language Curry which is intended to become a standard language in this area. It includes important ideas of existing functional logic languages and recent developments, and combines the most important features of functional and logic languages. Thus, Curry can be the basis to combine the currently separated research efforts of the functional and logic programming communities and to boost declarative programming in general. Moreover, since functions provide for more efficient evaluation strategies and are a declarative replacement of some impure features of Prolog (in particular, pruning operators), Curry can be also used as a declarative successor of Prolog.

1 Motivation

During the last decade, many proposals have been made to combine the most important declarative programming paradigms (see [15] for a survey). Functional logic languages offer features from functional programming (reduction of nested expressions, higher-order functions) and logic programming (logical variables, partial data structures, search for solutions). Compared to pure functional languages, functional logic languages have more expressive power due to the use of logical variables and built-in search mechanisms. Compared to pure logic languages, functional logic languages have more efficient evaluation mechanisms due to the (deterministic!) reduction of functional expressions (see [8, 13, 17] for discussions about the efficiency improvements of functional logic languages in comparison to Prolog). Thus, impure features of Prolog to restrict the search space, like the cut operator, can be avoided in functional logic languages. However, there is no obvious way to combine the search facilities of logic programming with efficient evaluation principles of functional programming. Functional approaches (i.e., (lazy) lists of successes [39]) require a directed data flow and do not allow partially instantiated data structures. Approaches which allow an arbitrary data flow have a tradeoff between completeness and efficiency (see discussion below on residuation and narrowing). As a consequence, quite different methods to integrate functional and logic languages have been proposed in the past. The most promising operational principles are residuation and narrowing.

*Informatik II, RWTH Aachen, D-52056 Aachen, Germany, {hanus,herbert}@informatik.rwth-aachen.de
†Departamento LSIIS, Facultad de Informática, Boadilla del Monte, 28660 Madrid, Spain, jjmoreno@fi.upm.es

Residuation is based on the idea to delay function calls until they are ready for deterministic evaluation. The residuation principle is used, for instance, in the languages Escher [22, 23], Le Fun [2], Life [1], NUE-Prolog [32], and Oz [38]. Since the residuation principle evaluates function calls by deterministic reduction steps, nondeterministic search must be explicitly encoded by predicates [1, 2, 32] or disjunctions [37]. The residuation principle is a reasonable integration of the functional and the logic paradigm since it combines the deterministic reduction of functions with partial data structures (logical variables). Moreover, it allows concurrent computation with synchronization on logical variables. However, it has also two disadvantages. Firstly, it is incomplete, i.e., it is unable to compute solutions if arguments of functions are not sufficiently instantiated during the computation. Secondly, it is not clear whether this strategy is better than Prolog's resolution strategy since there are examples where residuation has an infinite search space whereas the equivalent (flattened) Prolog program has a finite search space [16].

Functional logic languages with a complete operational semantics, e.g., ALF [12], Babel [28], K-Leaf [9], LPG [6], SLOG [8], are mainly based on *narrowing*, a combination of the reduction principle of functional languages with unification for parameter passing. Narrowing provides completeness in the sense of functional programming (normal forms are computed if they exist) as well as logic programming (solutions are computed if they exist). However, in order to compete with Prolog's resolution strategy, sophisticated *narrowing strategies* are required. *Innermost* or *eager narrowing* is equivalent to Prolog's left-to-right strategy if function calls are flattened into predicates. However, nested functional expressions allow the application of deterministic reduction steps between nondeterministic narrowing steps. Since such *normalizing narrowing* strategies can largely reduce the search space in comparison to pure logic programs, they form the basis of languages like ALF [12], LPG [6], or SLOG [8]. Since many modern functional languages are based on lazy evaluation, most recent work has concentrated on *lazy narrowing* strategies [7, 9, 26, 28, 36]. Similarly to lazy evaluation in functional languages, lazy narrowing evaluates an inner term only when its value is demanded to narrow an outer term. Thus, lazy narrowing avoids unnecessary computations of inner subterms and supports typical functional programming techniques like infinite data structures. In contrast to functional languages, a naive version of lazy narrowing may evaluate the same argument several times and may run into infinite loops (in contrast to eager narrowing!) due to the nondeterministic choice of a function's rewrite rules. Therefore, several methods have been proposed aiming at evaluating arguments commonly demanded by all rules before the nondeterministic choice [5, 11, 24, 27]. Among these different lazy narrowing strategies, there is one, called *needed narrowing* [5], which is optimal w.r.t. the length of derivations and the number of computed solutions. This clearly shows the advantages of integrating functions into logic programs: by transferring results from functional programming to logic programming, we obtain better and, for particular classes of programs, optimal evaluation strategies without loosing the search facilities. Defining functions is not a burden to the programmer since most predicates of application programs are functions. Moreover, the knowledge about functional dependencies can avoid useless computations (of arguments which are not needed) and increase the number of deterministic evaluation steps.

Improving the evaluation strategy is also a topic in logic programming [33]. However, most of the proposals are ad hoc ("cut") or do not exploit the full power of deterministic evaluations (Andorra model).[1] Therefore, functional logic languages improve logic languages by avoiding impure control features. Hence, functions are a declarative notion to improve control in logic programming. Moreover, they provide for useful functional programming techniques and lead to clearer programs.

[1]The Andorra computation model [18] prefers the evaluation of literals where at most one clause is applicable. In case of clauses with overlapping left-hand sides, there may be several clauses applicable leading to the same result. The possible pruning of the computation space in these cases is not covered by the Andorra model.

The currently existing functional logic languages cover only particular aspects of known results in this area and modern functional and logic languages in general. Therefore, the main motivation for Curry is to provide an integrated functional logic programming language which covers all important aspects of modern functional as well as logic languages. It should combine the best ideas of existing declarative languages, including

1. Haskell [20] and SML [25] (functional languages)
2. Gödel [19] and λProlog [30] (logic languages)
3. ALF [12], Babel [28], and Escher [22, 23] (functional logic languages)

Curry does not subsume each of these language but combines important aspects of them in a practical and comprehensive way. In the following, we will describe the functional logic language Curry which is based on the ideas described above. In the next section we will outline the operational semantics. In Section 3 we sketch the other important features of the language.

2 Operational Semantics

As discussed in the previous section, there is no clear view about the best operational semantics of functional logic languages. Residuation allows the efficient deterministic evaluation of function calls and provides for concurrent programming techniques, whereas narrowing is the basis of a complete and, for inductively sequential programs [5], optimal evaluation strategy but requires the implementation of search features. Although search can be costly and problematic in conjunction with I/O operations, it is one of the important extensions of pure functional programming. Therefore, Curry is based on a combination of narrowing and residuation. If the user does not specify any evaluation strategy, Curry chooses a strategy which is complete in the sense of functional and logic programming:

1. If there exists a solution to a goal, this solution (or a more general one) is computed.[2]
2. If an expression is reducible to some value (data term), Curry computes this value.[3]

In order to satisfy these requirements, Curry applies a sophisticated lazy narrowing strategy [5, 14, 24, 27]. However, if the programmer prefers another strategy, he can annotate functions with *evaluation restrictions*.[4] These evaluation restrictions specify that a function will not be evaluated until the arguments have a particular form. For instance, consider the concatenation on lists defined by[5]

```
function append: [A] -> [A] -> [A]
append []     L = L
```

[2]In order to implement Curry efficiently on sequential architectures, Curry implements search by backtracking which may cause incompleteness in the Prolog sense. However, the user is free to choose a breadth-first search strategy by particular search operators (see below).

[3]Ground terms based on functions defined by unconditional rewrite rules are evaluated in a fully deterministic way. However, if functions are defined by conditional rules with extra variables in conditions, some search may be necessary in order to apply such reduction rules. In this case, completeness depends on the completeness of the search strategy.

[4]Evaluation restrictions are comparable to coroutining declarations [31] in Prolog where the programmer specifies conditions under which a literal is ready for a resolution step. Moreover, they describe the strategy to evaluate different and nested arguments.

[5]The list notation is similar to Prolog. The type `[A]` denotes all lists with elements of type `A`.

```
append [E|R] L = [E | append R L]
```

Without any evaluation restrictions, Curry computes the answer `L=[1,2]` to the goal equation `append L [3,4] == [1,2,3,4]` by narrowing. However, if the evaluation restriction

```
eval append 1:rigid
```

is added, an `append` call is only reduced if the first argument is not headed by a defined function symbol and different from a logical variable,[6] otherwise the call is delayed. Such evaluation restrictions are implicit in Le Fun [2] and Life [1], explicit in Escher [22, 23], and automatically generated in NUE-Prolog [32]. Using evaluation restrictions, the programmer can specify any evaluation strategy between lazy narrowing and residuation. In this case the programmer is responsible to ensure that solutions can be computed even with the restricted evaluation strategy. On the other hand, there are program analysis methods which provide sufficient criteria to ensure the completeness of residuation [16].

In contrast to logic programming, functional logic programs contain nested function calls. Furthermore, the evaluation of some arguments is necessary only if some other arguments are evaluated to particular values. This is demonstrated by the following definition of the less-or-equal predicate on natural numbers represented by terms built from `0` and `s`:

```
function leq: nat -> nat -> bool
leq 0     N     = true
leq (s M) 0     = false
leq (s M) (s N) = leq M N
```

Consider a function call like (`leq` e_1 e_2). In order to apply some reduction rule, the first argument e_1 must always be evaluated to head-normal form (i.e., to a term without a defined function symbol at the top). However, the second argument must be evaluated only if the first argument has the form (`s` e).[7] This dependency between the first and the second argument can be expressed by the evaluation restriction

```
eval leq 1:(s => 2)
```

which specifies that the first argument is evaluated at the beginning and the second argument is only evaluated if the first argument has the constructor `s` at the top. In the general case, we can also specify deeper positions in arguments and nondeterministic selection of arguments in case of overlapping rules. Thus, evaluation restrictions uses definitional trees [4] and its generalizations [17, 24] which have been shown useful to specify sophisticated evaluation strategies for functional logic programs. Moreover, they can be mixed with information to specify that rules should not be applied if there is a logical variable at some argument position. For instance, the evaluation restriction

```
eval leq 1:rigid(s => 2:rigid)
```

specifies the obvious residuation strategy for `leq`.

In order to support eager evaluation strategies where arguments are reduced to normal form instead of head normal form, we also permit the annotation `nf` (similar to `rigid`). One possibility is to generate such annotations automatically by a "demandedness" analyzer.[8] Other reasonable extensions of evaluation restrictions are cyclic patterns to specify refined evaluation strategies [27].

[6]For arguments of functional type, `rigid` also requires that it is not of the form `F` $e_1 \ldots e_n$, i.e., a (partial) application of an unknown function.

[7]Naive lazy narrowing strategies may also evaluate the second argument in any case. However, as shown in [5], the consideration of dependencies between arguments is essential to obtain optimal evaluation strategies.

[8]In contrast to functional languages, strictness is not sufficient to safely replace lazy by (more efficient) eager evaluation in functional logic languages.

The general form of evaluation restrictions is defined in Appendix A.

3 Language Features

In this section we discuss various features of Curry.

3.1 Type System

Modern functional languages (e.g., Haskell [20], SML [25]) allow the detection of many programming errors at compile time by the use of polymorphic type systems. Similar type systems are also used in modern logic languages (e.g., Gödel [19], λProlog [30]). Curry has a polymorphic type system similar to Haskell, including type classes. Since Curry is a higher-order language, function types are written in their curried form τ_1 `->` τ_2 `->` $\cdots$ `->` τ_n `->` τ where τ is not a functional type. In this case, n is called the *arity* of the function.

Curry distinguishes between functions to construct data types, called *constructors*, and *defined functions* operating on these data types. Constructors are introduced by *data type declarations* like

```
datatype bool = true | false
datatype nat = 0 | s nat
datatype tree A = leaf A | node (tree A) A (tree A)
```

The extension of this type system to Haskell's type classes is a topic for future work.

3.2 Function Declarations

Functions are defined by a type declaration of the form

`function` f`:`τ_1 `->` τ_2 `->` $\cdots$ `->` τ_n `->` τ

where $\tau_1, \ldots, \tau_n, \tau$ are polymorphic types and τ is not a functional type, followed by conditional equations of the form

$f\ t_1 \ldots\ t_n$ `=` t `<=` C

where the conditional part "`<=` C" can be omitted. The left-hand side consists of the function symbol applied to a sequence of n patterns (i.e. variables or (full) applications of constructors to patterns). Note that defining rules of higher-type, e.g., `f = g` if `f` and `g` are of type `nat -> nat`, are excluded since this would cause a gap between the standard notion of higher-order rewriting and the corresponding equational theory [34]. Therefore, an equation `f = g` between functions is interpreted in Curry as syntactic sugar for the corresponding equation `f X = g X` on base types.

The condition C (also sometimes called a *goal*) is a conjunction of Boolean expressions and *strict equations* of the form l==r. A strict equation is provable if the left- and right-hand side are reducible to unifiable constructor terms.[9] Note that strict equality is the only sensible notion of equality in the presence of nonterminating functions [9, 28]. A Boolean expression is built from Boolean functions, predefined Boolean operators like "`,`" (and), "`;`" (or) and `not`[28]. `not` changes `true` to `false` and vice versa; it is *not* handled by finite failure.

In order to ensure the well-definedness and determinism of a function specified by several equations, additional non-ambiguity requirements are necessary (see [28] for details). In contrast to functional languages, we allow extra variables in the conditions, i.e., variables which do not occur

[9] In the theoretical setting, a strict equation is provable only if both sides are reducible to the same ground constructor term. Since goal variables are only instantiated to constructor terms, we can delay the ground instantiation of variables by unifying both sides which permits to deal with partial data structures as in Prolog (see [10, 24] for a more detailed discussion on this subject).

in the left-hand side. These extra variables provide the power of logic programming since a search for appropriate values is necessary in order to apply a conditional rule with extra variables.

Note that Curry has no special notation for predicates since they can be defined as Boolean functions. Facts and rules are represented by the defining equations

$p\ t_1 \ldots t_n$ `= true`
$p\ t_1 \ldots t_n$ `= true <=` $p_1\ s_{11} \ldots s_{1n_1}$ `, ... ,` $p_k\ s_{k1} \ldots s_{kn_k}$

The functional notation of predicates provides for more deterministic evaluations than the relational form.

3.3 Higher-order Features

Curry is a higher-order language supporting the common functional programming techniques by partial function applications and lambda abstractions. For instance, the well-known `map` function is defined in Curry by

```
function map: (A -> B) -> [A] -> [B]
map F []    = []
map F [E|L] = [(F E)|map F L]
```

However, there is an important difference to functional programming. Since Curry is also a logic language, it allows logical variables also for functional values, i.e., it is possible to evaluate the goal equation `map F [1 2] == [2 3]` which has, for instance, a solution `F=inc` if `inc` is the increment function on natural numbers. There are different proposals to deal with higher-order logical variables. In general, *higher-order unification* is necessary to compute all solutions to such goals [30, 35]. If logical variables at function positions are quantified over all (partial applications of) defined functions instead of all lambda expressions, higher-order unification can be avoided and replaced by an enumeration of all (type-conform) function symbols [10, 40]. A third alternative is to delay the application of unknown functions until the function becomes known [2, 38]. This last alternative can be implemented by residuation using the following special apply function:

```
function applyIfKnown: (A -> B) -> A -> B
eval applyIfKnown 1:rigid
applyIfKnown F A = (F A)
```

Thus, Curry supports only the first and second alternative. Curry provides a restricted form of higher-order unification (since the left-hand sides of function definitions are required to be patterns, in contrast to λProlog [30]) and an annotation for function variables specifying that these variables are quantified only over all function symbols occurring in the program.

Lambda terms are a useful data structure to capture the notion of bound variables and provide a comfortable way to manipulate programs as objects [30]. Lambda terms in left-hand sides of defining rules can be used to manipulate objects with bound variables and to capture the notion of scope in the object language. The following example contains a few rules of a symbolic differentiation function where Curry's abbreviation for equations of higher-order type is used (cf. Section 3.2):

```
function diff: (real -> real) -> real -> real
diff λX.F = λX.0
diff λX.X = λX.1
diff λX.(sin (F X)) = λX.(cos F X) * diff (λY.F Y) X
```

In the first rule, the variable `F` denotes an arbitrary function which does not depend on X (otherwise, the argument must have the form `λX.F(X)`). Therefore, `λX.F` matches only lambda abstractions where the body has no occurrence of the parameter, i.e., λ`X.F` matches only constant functions. Note that higher-order unification is necessary to correctly treat bound variables.

3.4 Encapsulated Search

Global search, possibly implemented by backtracking, must be avoided in some situations (user-control of efficiency, concurrent computations, non-backtrackable I/O). Hence it is sometimes necessary to encapsulate search in parts of larger programs. Search can take place in Curry whenever an argument must be evaluated with a logical variable as its actual value. In this case, the computation must follow different branches with different substitutions applied to the current goal. To give the programmer control on the actions taken in this situation, Curry provides a search operator similar to that of Oz [37]. However, in Curry it is not necessary to define a function by disjunctions in order to apply the search operator. Thus, the encapsulation of search is initiated by the caller and not visible in the definition of the called functions. This has the advantage that the same function can be used for deterministic evaluation or search depending on the structure of the actual arguments (ground terms or free variables).

Since search is used to find solutions to some Boolean expression, search is always initiated by some goal containing a *search variable* for which a solution should be computed.[10] Since the search variable may be bound to different solutions in different search paths, they must be abstracted. Therefore, a search goal has the form $\lambda x.g$ where x is the search variable contained in the goal g. To describe the result of the search steps, Curry offers a predefined data type

```
datatype searchspace A = failed | solved (A -> bool) | distributed [A -> bool]
```

Intuitively, `failed` represents a failed search, `solved` $\lambda x.g$ denotes a successful search where g is a satisfiable goal, and `distributed [`$\lambda x.g_1$`,...,`$\lambda x.g_n$`]` represents an intermediate search state. `distributed [`$\lambda x.g_1$`,...,`$\lambda x.g_n$`]` can be understood as a disjunction of goals. Moreover, there is a predefined function

```
function solve: (A -> bool) -> searchspace A
```

where `solve` $\lambda x.g$ evaluates the goal until it is not further reducible and unsatisfiable (in this case the result is `failed`), it is not further reducible but satisfiable (in this case the result is `solved` $\lambda x.g'$ representing the simplified goal), or it can be reduced to n different goals $\lambda x.g_1,\ldots,\lambda x.g_n$ by a nondeterministic narrowing step, i.e., there are at least n different rules applicable to the goal (in this case the result is `distributed [`$\lambda x.g_1$`,...,`$\lambda x.g_n$`]`). Thus, `solve` evaluates a goal at most until the first nondeterministic step occurs. In this case, it exposes an intermediate state of the search to the user, who can decide in which direction the search space should be explored further. For instance, the result of `solve λL.(append L [] == [0])` is

```
distributed [λL.(L==[],[]==[0]), λL.∃X∃L1(L==[X|L1],[X|append L1 []]==[0])]
```

(Curry also provides existential quantifiers, see Section 3.7). To avoid conflicting variable bindings caused by distributed goals, `solve` requires an argument without free variables. A depth-first search strategy can be formulated as:

```
function depthfirst: [A -> bool] -> searchspace A
depthfirst []     =  failed
depthfirst [X|Xs] =
    case  solve X  of
       failed:     depthfirst Xs
       solved Y:   solved Y
```

[10] The generalization to more than one search variable is straightforward by using tuples.

```
distributed [Z|Zs]:
   case  depthfirst [Z]  of
      solved V:  solved V
      failed:    depthfirst (append Zs Xs)
```

Besides depth-first search, which computes only the leftmost solution, many other kinds of search strategies can be specified, including breadth-first search, collecting all solutions in a list, etc. A library of typical search strategies is provided, such that the casual user does not have to bother on how to implement them.

The search operators can be used in top-level goals as well as in conditions of rules. For instance, we can compute by `(depthfirst [λL.(append L [1] == [0,1])]) == S` a solution of this goal. If `S` is of the form `solved G`, we can bind by the application `G X` a global variable `X` to the value `[0]`.

3.5 Monadic I/O

Curry provides a declarative model of I/O by considering I/O operations as transformations on the outside world. In order to avoid dealing with different versions of the outside world, it must be ensured that at each point of a computation only one version of the world is accessible. This ensured by using monadic I/O like in Haskell and by requiring that I/O operations are not allowed in program parts where nondeterministic search is possible. Thus, all search must be encapsulated between I/O operations. Using the evaluation restrictions, the compiler is able to detect functions where search is definitely avoided (if all evaluated positions are declared as `rigid`). In combination with search operators, the compiler can infer that search will not take place ensuring well-defined declarative I/O operations.

3.6 Constraints

The integration of predefined data types by the use of constraints has been shown useful in logic programming. Hence constraints are a necessary feature of any modern logic programming language. However, the combination of arbitrary constraints with sophisticated narrowing strategies is a topic for current and future research. As a consequence, the current version of Curry does not support arbitrary constraints. Curry provides disequality constraints [21] as a method to express negative information. The inclusion of other constraint systems like arithmetic constraints, finite domains, feature terms or record structures by a uniform interface is a topic for future extensions.

3.7 Implication and Quantifiers

Implication and quantifiers inside conditions are a useful feature of higher-order logic languages [29, 30] since they are a declarative alternative to some impure features of Prolog, in particular, `assert` and `retract`. Moreover, they provide scoping constructs in logic programming. Therefore, Curry supports these features. When evaluating an implication *rules* `=>` *e*, *rules* are added to the program while evaluating *e*. The value of the implication is the value of *e*. If the evaluation of *e* fails or is finished, *rules* are removed from the program.

The combination of quantifiers and implications provide for scoping constructs to improve the structure of larger programs. For instance, if a predicate `select` is only an auxiliary predicate to define the predicate `perm`, it should be made local to `perm`. This is possible by the use of quantifiers and implications:

```
perm([],[])
```

```
perm([E|L],[F|M]) <=
 (∀select(∀E∀L(select(E,[E|L],L)) ∧
        ∀E∀F∀L∀M(select(E,[F|L],[F|M]) <= select(E,L,M)))
  => select(F,[E|L],N), perm(N,M))
```

The scope of the name of the auxiliary predicate `select` is restricted by the quantifier `∀select` inside the second clause of `perm` (see [29] for more examples for scoping constructs). Of course, the user is not forced to use this awkward notation since Curry offers `where` clauses as syntactic sugar for the previous clauses:

```
perm([],[])
perm([E|L],[F|M]) <= select(F,[E|L],N), perm(N,M)
   where  select(E,[E|L],L)
          select(E,[F|L],[F|M]) <= select(E,L,M)
```

3.8 Modules

The design of a module system for Curry is not influenced by the functional logic features of Curry. Therefore, the current version of Curry uses a standard module system similar to ALF's [12] or Gödel's [19]. The extension to a more sophisticated module system like in SML [25] is a topic for future extensions.

4 Example

Let us consider an example program in order to demonstrate some features of Curry. We want to compute all homomorphisms between two abelian groups. The homomorphism condition is checked for all pairs of elements of the first group.

```
function hom: [nat] -> (nat->nat->nat)
      -> [nat] -> (nat->nat->nat) ->  (nat -> nat) -> bool
hom G1 Op1 G2 Op2 F =   and [ test Op1 Op2 F X Y | X <- G1, Y <- G1]
function test: (nat->nat->nat) -> (nat->nat->nat) ->  (nat -> nat)
      ->  nat -> nat -> bool
test Op1 Op2 F X Y  =  true  <=  Op2 (F X) (F Y) == F (Op1 X Y)
```

A valid query for the above program is

```
hom [0,1,2,3] add4 [0,1] add2 mod2
```

which would check, whether the remainder of the division by 2 (`mod2`) is a homomorphism between the groups $\langle\{0,1,2,3\},\ \texttt{add4}\rangle$ and $\langle\{0,1\},\ \texttt{add2}\rangle$, where `add4` and `add2` are the addition modulo 4 and modulo 2, respectively. `mod2`, `add4`, and `add2` as well as the conjunction `and` of list elements, are, among others, assumed to be predefined by appropriate rules. `[ test X Y | X <- G1, Y <- G1]` is a list comprehension (see e.g. [20]), and denotes the list of values `test X Y` where `X` and `Y` range over the elements of the first group `G1`.

The above goal can already be handled in "ordinary" functional (logic) languages. However, Curry allows a goal like the following, which requires (generalized) higher order unification:

```
hom [0,1,2,3] add4 [0,1] add2 F
```

Here, the variable `F` is bound to a homomorphism between the two groups. Thus, Curry allows to *search for functions*. A possible solution is `F=mod2`. Note that Curry also considers solutions which are composed of projections, constructor symbols *and defined functions* (in contrast to λProlog) (see [3] for more details).

5 Implementation

Although this paper describes only the design of Curry, we will also briefly discuss some implementation aspects. Curry combines very powerful concepts. However, Curry contains various restrictions that allow an efficient implementation by transferring known implementation techniques to Curry. The omission of defined function symbols in arguments of left-hand sides of function definitions provide for efficient evaluation strategies (see [15] for a survey of different strategies and the importance of constructor-based rules). The restriction to patterns in left-hand sides ensure that full higher-order unification is rarely used [35]. Moreover, the evaluation restrictions permit the definition of application-specific evaluation strategies. However, in this case the programmer is responsible to ensure the completeness of his strategy.

6 Conclusions

From the (informal) description of Curry in the previous sections it should be clear that Curry is a real integration of functional and logic languages since it covers most aspects of both paradigms. For functional programming, Curry provides higher-order functions, lazy evaluation and deterministic evaluation of ground expressions. Logic programming features are supported by logical variables, partial data structures and search facilities. It is interesting to note that each purely logic program can be simply mapped into a Curry program by mapping each clause `p :-` $\mathtt{p}_1$`,...,`$\mathtt{p}_n$ into the equation

`p=true <=` $\mathtt{p}_1$`,...,`$\mathtt{p}_n$

If the evaluation restriction of the conjunction "`,`" is `1:(true => 2)`, Curry's narrowing strategy is equivalent to Prolog's left-to-right resolution strategy. However, without any evaluation restrictions, Curry is free to choose a more sophisticated strategy which prefer deterministic evaluations in literals other than the leftmost one.

By the availability of several new features in comparison to pure logic programming, Curry avoids the following impure constructs of Prolog:

- The *cut* and similar pruning operators are replaced by the deterministic evaluation of functions (note that there is no direct replacement of the cut in Curry since deterministic function evaluation corresponds to "green cuts" due to the complete operational semantics of Curry).
- The *call predicate* is replaced by the higher-order features of Curry.
- Many applications of `assert` and `retract` can be eliminated using implications in conditions.
- The *I/O operations* of Prolog are replaced by the declarative monadic I/O concept of functional programming.

Although not every impure Prolog program can be directly mapped into a Curry program, we think that Curry is a suitable declarative alternative for most application problems written in Prolog.

The present proposal is far from its final shape. Its purpose is to stimulate the discussion on a standardized functional logic language. Many aspects have not yet been addressed and may have to be included, for instance, metaprogramming features, default rules, bounded quantification, constraints, a dedicated software environment (e.g. a debugger).

Acknowledgement

The authors would like to thank Sergio Antoy, John Lloyd, Rita Loogen, and Mario Rodríguez-Artalejo for a lot of valuable comments and suggestions on this paper.

References

[1] H. Aït-Kaci. An overview of LIFE. In J.W. Schmidt and A.A. Stogny, editors, *Proc. Workshop on Next Generation Information System Technology*, pages 42–58. Springer LNCS 504, 1990.

[2] H. Aït-Kaci, P. Lincoln, and R. Nasr. Le Fun: Logic, equations, and functions. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pages 17–23, San Francisco, 1987.

[3] J. Anastasiadis and H. Kuchen. Higher order babel: Language and implementation. In *Proc. Workshop of Extensions of Logic Programming*, 1996.

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

[5] 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.

[6] D. Bert and R. Echahed. Design and implementation of a generic, logic and functional programming language. In *Proc. European Symposium on Programming*, pages 119–132. Springer LNCS 213, 1986.

[7] J. Darlington and Y. Guo. Narrowing and unification in functional programming - an evaluation mechanism for absolute set abstraction. In *Proc. of the Conference on Rewriting Techniques and Applications*, pages 92–108. Springer LNCS 355, 1989.

[8] L. Fribourg. SLOG: A logic programming language interpreter based on clausal superposition and rewriting. In *Proc. IEEE Internat. Symposium on Logic Programming*, pages 172–184, Boston, 1985.

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

[10] J.C. González-Moreno, M.T. Hortalá-González, and M. Rodríguez-Artalejo. On the completeness of narrowing as the operational semantics of functional logic programming. In *Proc. CSL'92*, pages 216–230. Springer LNCS 702, 1992.

[11] W. Hans, R. Loogen, and S. Winkler. On the interaction of lazy evaluation and backtracking. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pages 355–369. Springer LNCS 631, 1992.

[12] M. Hanus. Compiling logic programs with equality. In *Proc. of the 2nd Int. Workshop on Programming Language Implementation and Logic Programming*, pages 387–401. Springer LNCS 456, 1990.

[13] M. Hanus. Improving control of logic programs by using functional logic languages. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pages 1–23. Springer LNCS 631, 1992.

[14] M. Hanus. Combining lazy narrowing and simplification. In *Proc. of the 6th International Symposium on Programming Language Implementation and Logic Programming*, pages 370–384. Springer LNCS 844, 1994.

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

[16] M. Hanus. Analysis of residuating logic programs. *Journal of Logic Programming*, 24(3):161–199, 1995.

[17] M. Hanus. Efficient translation of lazy functional logic programs into Prolog. In *Proc. of the Fifth International Workshop on Logic Program Synthesis and Transformation*, 1995.

[18] S. Haridi and P. Brand. Andorra prolog: An integration of prolog and committed choice languages. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, pages 745–754, 1988.

[19] P.M. Hill and J.W. Lloyd. *The Gödel Programming Language.* MIT Press, 1994.

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

[21] H. Kuchen, F.J. López-Fraguas, J.J. Moreno-Navarro, and M. Rodríguez-Artalejo. Implementing a lazy functional logic language with disequality constraints. In *Proc. of the 1992 Joint International Conference and Symposium on Logic Programming*. MIT Press, 1992.

[22] J.W. Lloyd. Combining functional and logic programming languages. In *Proc. of the International Logic Programming Symposium*, pages 43–57, 1994.

[23] J.W. Lloyd. Declarative programming in Escher. Technical report cstr-95-013, University of Bristol, 1995.

[24] 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.

[25] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[26] J.J. Moreno-Navarro, H. Kuchen, R. Loogen, and M. Rodríguez-Artalejo. Lazy narrowing in a graph machine. In *Proc. Second International Conference on Algebraic and Logic Programming*, pages 298–317. Springer LNCS 463, 1990.

[27] J.J. Moreno-Navarro, H. Kuchen, J. Marino-Carballo, S. Winkler, and W. Hans. Efficient lazy narrowing using demandedness analysis. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pages 167–183. Springer LNCS 714, 1993.

[28] 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.

[29] G. Nadathur, B. Jayaraman, and K. Kwon. Scoping constructs in logic programming: Implementation problems and their solution. *Journal of Logic Programming*, 25(2):119–161, 1995.

[30] G. Nadathur and D. Miller. An overview of λProlog. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pages 810–827. MIT Press, 1988.

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

[32] L. Naish. Adding equations to NU-Prolog. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pages 15–26. Springer LNCS 528, 1991.

[33] L. Naish. Pruning in logic programming. Technical report 95/16, University of Melbourne, 1995.

[34] T. Nipkow. Higher-order critical pairs. In *Proc. 6th IEEE Symp. Logic in Computer Science*, pages 342–349. IEEE Press, 1991.

[35] C. Prehofer. Higher-order narrowing. In *Proc. Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 507–516, 1994.

[36] U.S. Reddy. Narrowing as the operational semantics of functional languages. In *Proc. IEEE Internat. Symposium on Logic Programming*, pages 138–151, Boston, 1985.

[37] C. Schulte and G. Smolka. Encapsulated search for higher-order concurrent constraint programming. In *Proc. of the 1994 International Logic Programming Symposium*, pages 505–520. MIT Press, 1994.

[38] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Current Trends in Computer Science*. Springer LNCS 1000, 1995.

[39] P. Wadler. How to replace failure by a list of successes. In *Functional Programming and Computer Architecture*. Springer LNCS 201, 1985.

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

A Evaluation Restrictions

The general form of evaluation restrictions is "`eval` *f* *restriction*" where *restriction* is defined by the following grammar:

restriction	::=	*position* [`:`*annotation*]	% evaluate position
	|	*restriction* `or` *restriction*	% alternative argument evaluations
position	::=	*number*	
	|	*number*`.`*position*	
annotation	::=	*rigid* [`(`*crestriction**`)`]	% proceed if position is rigid
	|	*nf* [`(`*crestriction**`)`]	% compute normal form
	|	`(`*crestriction**`)`	% proceed in any case
crestriction	::=	*c* `=>` *restriction*	% *c* is a constructor