Functional Logic Programming

Sergio Antoy
Portland State University
Portland, OR 97207, U.S.A.
antoy@cs.pdx.edu

Michael Hanus
Institut für Informatik, CAU Kiel
D-24098 Kiel, Germany.
mh@informatik.uni-kiel.de

1. INTRODUCTION

The evolution of programming languages is the stepwise introduction of abstractions hiding the underlying computer hardware and the details of program execution. Assembly languages introduce mnemonic instructions and symbolic labels for hiding machine codes and addresses. Fortran introduces arrays and expressions in standard mathematical notation for hiding registers. Algol-like languages introduce structured statements for hiding *gotos* and jump labels. Object-oriented languages introduce visibility levels and encapsulation for hiding the representation of data and the management of memory. Along these lines, declarative languages, the most prominent representatives of which are functional and logic languages, hide the order of evaluation by removing assignment and other control statements. A declarative program is a set of logical statements describing properties of the application domain. The execution of a declarative program is the computation of the value(s) of an expression with respect to these properties. Thus, the programming effort shifts from encoding the steps for computing a result to structuring the application data and the relationships between the application components.

Declarative languages are similar to formal specification languages, but with a significant difference: they are *executable*. The language that describes the properties of the application domain is restricted to ensure the existence of an efficient evaluation strategy. Different formalisms lead to different classes of declarative languages. *Functional languages* are based on the notion of mathematical function; a functional program is a set of functions that operate on data structures and are defined by equations using case distinction and recursion. *Logic languages* are based on predicate logic; a logic program is a set of predicates defined by restricted forms of logic formulas, such as Horn clauses (implications).

Both kinds of languages have similar motivations but provide different features. E.g., functional laguages provide efficient, demand-driven evaluation strategies that support infinite structures, whereas logic languages provide non-determinism and predicates with multiple input/output modes that offer code reuse. Since all these features are useful for developing software, it is worthwhile to amalgamate them into a single language. *Functional logic languages* combine the features of both paradigms in a conservative manner. Programs that do not use the features of one paradigm behave as programs of the other paradigm. In addition to the convenience of using the features of both paradigms within a single language, the combination has additional advantages. For instance, the demand-driven evaluation of functional programming applied to non-deterministic operations of logic programming leads to more efficient search strategies. The effort to develop languages intended to meet this goal has produced a substantial amount of research spanning over two decades (see [22] for a recent survey). These achievements are reviewed in the following. The concrete syntax of the examples is Curry [27], but the design of the code and most of our considerations about the programs are valid for any other functional logic language, e.g., $\mathcal{TOY}$ [30], based on reduction for functional evaluation and on narrowing for the instantiation of unbound logic variables.

A common trait of functional and logic languages is the distinction between data constructors and defined operations. This distinction is also essential for functional logic languages to support reasonably efficient execution mechanisms. *Data constructors* build the data underlying an application, whereas *defined operations* manipulate these data. For instance, `True` and `False` construct (are) Boolean values. A simple example of a more complex data structure is a list of values. We denote by `[]` the empty list and by `(x:xs)` a non-empty list where `x` is the first element and `xs` is the list of remaining elements. We create longer lists by nested applications of constructors: `(1:(2:(3:[])))` denotes a list with elements `1`, `2`, and `3`. Since lists are ubiquitous structures in declarative languages, syntactic sugar is usually provided to ease writing lists; thus, `[1,2,3]` abbreviates the previous list structure.

A further feature common to many functional and logic languages is the definition of operations by *pattern matching*. Functions are specified by different equations for different argument values. For instance, a function, "`++`", returning the concatenation of two input lists is defined as follows (here we use infix notation for "`++`"):

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

The first equation is applicable to calls to "`++`" with an empty list as the first argument. The second equation is used to evaluate calls with non-empty lists. The pattern `(x:xs)` of

Comm. of the ACM | April 2010 | vol. 53 | no. 4.

the second equation combines case distinction (the list is not empty) and selectors (let `x` and `xs` be the head and tail of the argument list) in a compact way. The meaning of this definition is purely equational; expressions are evaluated by replacing left-hand sides by corresponding right-hand sides (after substituting the equation's variables with corresponding actual arguments). These replacement steps are also called *reductions*, and the defining equations are often called *rules*, since they are applied from left to right.

The definition of functions by equations containing patterns in the parameters of the left-hand sides is well known from functional languages, such as ML [32] or Haskell [34]. Purely functional languages stipulate that rules must be *constructor-based*. The patterns in each defining rule consist only of constructors and variables (formal parameters). This excludes rules like

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
```

Such an equation describes a property (associativity) of the operation "`++`" rather than a constructive definition about its evaluation. The restriction to constructor-based rules is an important factor to support execution with efficient strategies. "Execution" in functional languages means reducing expressions containing defined operations to *values* (i.e., expressions without defined operations) by applying defining equations ("replacing equals by equals").

The *logic* component of functional logic languages comes into play when computing with incomplete information. The problem is to evaluate an expression containing a subexpression e such that the value of e is unknown, but it is known that e must satisfy certain conditions. In a computation, e is represented by a free variable and the conditions on e constrain the values that e may assume. For instance, consider the equation `zs++[2] =:= [1,2]`. (We use the symbol "`=:=`" for equations that should be solved in order to syntactically distinguish them from equations that define operations.) This equation states that we are interested only in the values for the variable `zs` that satisfy the equation. In this case we have to replace or instantiate `zs` with the list `[1]`. The combination of variable instantiation and term reduction is called *narrowing*, originally introduced in automated theorem proving [37] and first proposed for programming in [35]. In general, there might be infinitely many instantiations for the free variables of an expression. The research on functional logic languages has led to reasonable narrowing strategies that avoid a blind guessing of all the potential values of a variable [4]. Good strategies, as discussed later, perform only "constructive" guessing steps: They instantiate free variables only if the instantiation is necessary to sustain a computation and only with those values that are demanded to perform a reduction step. For instance, in the evaluation of `zs++[2]`, `zs` is replaced either by the empty list `[]` or by a non-empty list `(x:xs)` where the head and tail are again unknown. Either instantiation enables a reduction step with one of the rule defining "`++`". Not every instantiation will lead to a result; thus, some instantiation might be later discarded. Good narrowing strategies ensure that the cost of an instantiation is incurred only when a guess for an unknown value is necessary. Expressions without unknowns or with unknowns that don't need to be known are evaluated as in functional languages.

The capability to compute with unknowns supports new forms of code reuse. As we have seen in the equation above, we can compute the prefix of a list by narrowing. Similarly, the equation `zs++[e] =:= [1,2,3]` is solved by instantiating `e` with the last element of the right-hand side list (and `zs` with the remaining list). Thus, we can reuse the concatenation operation "`++`" to compute the last element of a list. We can also define an explicit function `last` for this purpose:

```
last xs | zs++[e] =:= xs
        = e
        where zs,e free
```

Here, we add both a *condition* to a defining rule, so that this rule is applicable only if the condition (the equation between "`|`" and "`=`") can be solved, and a declaration of the variables introduced by the rule (the `where ... free` clause).

Although code reuse originating from the evaluation of operations with unknown arguments is well known in logic programming, functional logic languages provide additional structures for reusing code. For instance, it is apparent from the above rule defining `last` that this rule is applicable only if the actual argument has a form that matches the result of narrowing `zs++[e]`. Thus, we can re-formulate the above rule as:

```
last (zs++[e]) = e
```

Note that purely functional languages, such as Haskell, do not allow this rule because it is not constructor-based; rather it contains a *functional pattern*, that is, a pattern with a defined function inside. When a rule of this kind is applied to some function call, the functional pattern is evaluated (by narrowing) to some value (which likely contains variables) which is then unified with the actual argument. Since the functional pattern `zs++[e]` can be narrowed to infinitely many values (`[e]` for `zs`=`[]`, `[x1,e]` for `zs`=`[x1]`, ...), it abbreviates an infinite number of ordinary patterns. Similarly to narrowing, it is not necessary to guess all these patterns at run time. Instead, the necessary patterns can be computed in a demand-driven manner during pattern matching [7].

These examples show the potential of functional logic languages: writing programs as clear specifications that abstract from the evaluation order, contain unknowns, and reuse defined operations in various ways. Of course, there is a price to pay for these advanced features. Dealing with unknowns requires the consideration of different alternatives during run time; in other words, computations might be non-deterministic. This non-determinism is *don't-know*, and finding all the solutions of a problem may require considering many non-deterministic alternatives. To make nondeterminism practically useful, the number of alternatives that must be considered by the execution of a program should be contained. This is provided by the evaluation strategy that will be the subject of Section 3.

2. CURRY

Curry [27] is a functional logic language developed by an international community of researchers to produce a standard for research, teaching, and application of functional logic programming. Details can be found at `www.curry-language.org`. In the following we give an overview of Curry with emphasis on aspects relevant to functional logic programming.

The syntax of Curry borrows heavily from that of Haskell

[34]. In fact, Curry mainly introduces a single syntactic extension (the declaration of free variables) with respect to Haskell, although the underlying evaluation strategy is different (see below). Thus, a Curry *program* is a set of definitions of data types and operations on values of these types. A *data type* is declared by enumerating all its constructors with the respective argument types. For example:

```
data Bool = True | False
data BTree a = Leaf a
             | Branch (BTree a) (BTree a)
```

The type `BTree` is *polymorphic*, that is, the type variable `a` ranges over all possible type expressions, and it has two constructors `Leaf` and `Branch`. Operations on polymorphic data structures are often generic. For instance, the following operation computes the number of nodes (branches and leaves) of a binary tree, where the (optional) first line defines the type signature:

```
size :: BTree a -> Int
size (Leaf _) = 1
size (Branch t1 t2) = 1 + size t1 + size t2
```

Thus, `size (Branch (Leaf 1) (Leaf 3))` evaluates to `3`. As in Haskell, the names of (type) variables and operations usually start with lowercase letters, whereas the names of type and data constructors start with an uppercase letter. The application of f to e is denoted by juxtaposition ($f\ e$), except for infix operators such as "`+`". Curry, in contrast to Haskell, may use the operation `size` to compute binary trees of a particular size. For example, if `t` is a free variable, the evaluation of `size t =:= 3` instantiates `t` to the tree structure `(Branch (Leaf x1) (Leaf x2))`, where `x1` and `x2` are free variables that remain unspecified because their values do not affect the equation.

As discussed earlier, finding solutions to an under-specified problem requires the non-deterministic evaluation of some expression. The ability to perform non-deterministic computations supports an interesting language feature, namely the definition of *non-deterministic operations*. These operations deliver more than one result with the intended meaning that one result is as good as any other. The archetype of non-deterministic operations is the binary infix operation "`?`", called *choice*, that returns one of its arguments:

```
x ? y = x
x ? y = y
```

Thus, we can define the flipping of a coin as:

```
coin = 0 ? 1
```

Non-deterministic operations are unusual at first glance, but they are quite useful for programming. Note that the result of a non-deterministic operation is a single (indeterminate) value rather than the set of all possible values. Thus, the value of `coin` is not the set $\{0, 1\}$, but one element of this set. The "single-element" view is important to exploit demand-driven evaluation strategies for the efficient evaluation of such operations. For instance, it might not be necessary to compute all results of a non-deterministic operation if the context demands only values of a particular shape. A concrete example of this advantage is given in the following paragraphs. It should be noted that there exists a declarative foundation (i.e., model-theoretic, big-step, and fixpoint semantics) for non-deterministic operations [19].

To show a slightly more interesting example involving non-deterministic operations, consider the definition of an operation that inserts an element into a list at an indeterminate position:

```
insert x ys     = x : ys
insert x (y:ys) = y : insert x ys
```

Since both rules are applicable in evaluating a call to `insert` with a non-empty list, `insert 0 [1,2]` evaluates to any of `[0,1,2]`, `[1,0,2]`, or `[1,2,0]`. Thus, `insert` supports a straightforward definition of a permutation of a list by inserting the first element at some position of a permutation of the tail of the list:

```
perm []     = []
perm (x:xs) = insert x (perm xs)
```

Permutations are useful for specifying properties of other operations. For instance, a property of a sort operation on lists is that the result of sorting is a permutation of the input list where all the elements are in ascending order. Since functional logic languages can deal with several results of an operation as well as failing alternatives, it is reasonable to specify sorted lists by an operation `sorted` that is the identity only on sorted lists (and fails on lists that are not sorted):

```
sorted []       = []
sorted [x]      = [x]
sorted (x:y:ys) | x<=y = x : sorted(y:ys)
```

Altogether, the expression "`sorted (perm xs)`" specifies the sorted permutation of a list `xs`. Since Curry evaluates non-deterministic definitions, this specification of sorting is executable. Although it seems that a complete strategy has to compute *all* the permutations, a good strategy does better than that. Modern functional logic languages, including Curry, use demand-driven strategies that do not always evaluate expressions completely. For instance, the argument `(perm xs)` of the expression `sorted (perm xs)` is evaluated only as demanded by `sorted`. If a permutation starts with out of order elements, as in `2:1:perm ys`, `sorted` will fail on this expression without further evaluating `perm ys`, i.e., the evaluation of all the permutations of `ys` is avoided. This demand-driven search strategy reduces the overall complexity of the computation compared to a simple generate-and-test strategy. Compared to other sorting algorithms, this program is still inefficient since we have not used specific knowledge about sorting collections of objects efficiently.

Non-deterministic operations promote concise definitions, as shown by `perm` above. It is interesting to observe that narrowing, that is, computation with free variables that are non-deterministically instantiated, and the evaluation of non-deterministic operations without free variables, have the same expressive power. For instance, one can replace a free variable of type `Bool` in an expression by the non-deterministic operation `genBool` that is defined by

```
genBool = True ? False
```

so that it evaluates to any value of type `Bool`. The equivalence of narrowing with free variables and the evaluation of such non-deterministic generator functions is formally stated in [8] and the basis of a recent Curry implementation [13].

Nevertheless, modern functional logic languages provide both features because both are convenient for programming. There is a subtle aspect to consider when non-deterministic expressions are passed as arguments to operations. Consider

the operation:

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

and the expression "`double coin`". Evaluating the argument `coin` (to `0` or `1`) before passing it to `double` yields `0` and `2` as results. If the argument `coin` is passed unevaluated to `double`, we obtain in one rewrite step the expression `coin+coin` which has four possible rewrite derivations resulting in the values `0`, `1` (twice), and `2`. The former behavior is referred to as *call-time choice* semantics [28] since the choice of the value of a non-deterministic argument is made at call time, whereas the latter is referred to as *need-time choice* semantics since the choice is made only when needed.

Although the call-time choice resembles an eager or call-by-value evaluation behavior, it fits well into the framework of demand-driven or lazy evaluation where arguments are shared to avoid the repeated evaluation of the same expression. For instance, the actual argument (e.g., `coin`) associated to the formal parameter `x` in the rule "`double x = x+x`" is not duplicated in the right-hand side. Rather both occurrences of `x` refer to the same term, which consequently is evaluated only once. This technique, called *sharing*, is essential to obtain efficient (and optimal) evaluation strategies. The call-time choice is the semantics usually adopted by current functional logic languages since it satisfies the principle of "least astonishment" in most situations [19].

Curry also supports the use of constraint solvers, since the condition of a rule is not restricted to a Boolean expression as in Haskell. The condition of a rule can be any *constraint* that must be solved before applying the rule. A constraint is any expression of the predefined type `Success`. The type `Success` is a type without constructors but with a few basic constraints that can be combined into larger expressions. We have already seen the constraint operation "`=:=`", which is a function that maps its arguments into the type `Success`. Furthermore, there is a conjunction operation on constraints, "`&`", that evaluates its two arguments concurrently. Beyond these basic constraints, some Curry implementations also offer more general constraint structures, such as arithmetic constraints on real numbers or finite domain constraints, together with appropriate constraint solvers. This enables the implementation of very effective algorithms to solve specific constraints. In this way, programs access and combine predefined constraints in a high-level manner.

Curry has a variety of other language features for high-level general purpose programming. Similarly to Haskell, it is strongly typed with a polymorphic type system for reliable programming. Generic programming is also obtained through higher-order operations. Curry supports modules and offers input/output using the monadic approach like Haskell [34]. Moreover, it predefines primitive operations to encapsulate search, that is, to embed the non-deterministic search for solutions into purely functional computations by passing some or all the solutions into list structures. The combined functional and logic programming features of Curry have been shown useful in diverse applications: for example, to provide high-level APIs to access and manipulate databases [12], to construct graphical user interfaces [26], and to support web programming [21, 26]. These developments were instrumental in identifying new design patterns that are specific to functional logic programming [6].

3. STRATEGY

A crucial semantic component of a functional logic programming language is its evaluation strategy. Informally, the strategy tells which subexpression of an expression should be evaluated first. For non-strict semantics, such demand-driven semantics, that do not evaluate every subexpression, the problem is non-trivial. For functional logic languages the inherent difficulties of the problem are compounded by the presence of incomplete information in the form of free variables. The formalization of an evaluation strategy requires a formal model of computation. Various models have been proposed for functional logic programming. Rewriting systems [9] are the model that more than any other has promoted significant progress in this area.

A functional logic *computation* is the repeated transformation, by narrowing steps, of an expression e. An *elementary step* of e involves a subexpression of e. This subexpression either is a redex (*red*ucible *ex*pression) or is instantiated to obtain a redex. This redex is then reduced according to some rewrite rule of the program. The strategy produces the subexpression, the optional instantiation, and the rewrite rule that constitute a step. The strategy may produce several elementary steps of an expression. Steps that do not "interfere" with each other can be combined into a *multistep* and executed concurrently.

A computation of an expression *terminates* when the expression does not allow further steps. This expression is called a *normal form*. If a normal form contains occurrences of defined operations, such as a division by zero or the head of an empty list, it is called a *failure*, and the corresponding computation is said to *fail*. Otherwise, the normal form is called a *result* or *value* and the corresponding computation is said to *succeed*. The non-deterministic nature of functional logic programming may lead to multiple outcomes of a computation. An expression may have several distinct results and several distinct failures. Failures are discarded, but failing computations may be both desirable and explicitly planned in the design of some programs [6]. The primary task of the strategy is to ensure that every result of an expression is eventually obtained. Not surprisingly, for functional logic programs it is undecidable to tell whether a computation will terminate.

The strategy of contemporary implementations of functional logic languages is based on a formalism called a definitional tree. A *definitional tree* [2] is a hierarchical structure of the rewrite rules defining an operation of a program. For example, consider the operation that returns a prefix of a given length of a list, where for the purpose of illustration natural numbers are represented by `Zero` and `Successor`.

```
take Z     _      = []
take (S n) []     = []
take (S n) (x:xs) = x : take n xs
```

The operation `take` makes an initial distinction on its first argument. The cases on this argument are zero and non-zero. In the non-zero case, the operation makes a subsequent distinction on its second argument. The cases on this argument are null and non-null. A definitional tree of the operation `take` [4, Example 8] encodes these distinctions, the order in which they are made, and the cases that they consider.

Definitional trees guide the evaluation strategy similarly to case expressions of functional languages, but there are significant differences. Case expressions are explicitly coded

by the programmer, whereas definitional trees are inferred from the rules defining an operation. This difference becomes apparent in some situation where both strategies are applicable. For example, consider the following operation:

```
f 0 0 = 0
f _ 1 = 1
```

In Curry, the computation of `f` t `1` results in `1` even if t does not terminate, whereas in the lazy functional language Haskell the same computation does not terminate on non-terminating t due to the fixed left-to-right evaluation of demanded arguments.

To fully support non-determinism, in a functional logic program the textual order of the rules defining an operation is irrelevant. A consequence of this stipulation is that not every operation has a definitional tree, and definitional trees may have different characteristics. This has prompted a classification of functional logic programs according to the existence and/or the kinds of the definitional trees of their operations. Implementations of functional logic languages have grown more sophisticated over time. The modern approach transforms a program of a source language, which allows operations without a definitional tree, functional patterns, and/or partially applied functions, into a semantically equivalent program of a core language in which every operation has a definitional tree [3, 8]. Each such tree is compiled into code or bytecode that implements a finite state automaton that determines both the evaluation of the arguments of a function application and the instantiation of free variables when necessary to perform a reduction.

Essential properties of an evaluation strategy are *soundness* (each computed result is correct with respect to an equational model of the program) and *completeness* (for each correct result, a corresponding value or a representative of this value is computed). Sound and complete strategies support a high-level abstract view of programming: The programmer states the intended properties of the application domain rather than the effects of individual computation steps.

Very strong properties are known for the evaluation strategy of Curry's core language. In particular, *needed narrowing* [5] is sound and complete, computes only needed steps (each computation has minimal length), and computes only disjoint instantiations (no computation is unnecessarily repeated).

The last two decades have seen a wealth of results in the area of strategies. Despite these achievements, we believe that more remains to be discovered. The promise lies in the development of strategies able to exploit the potential of parallel architectures. We will briefly discuss some possibilities in Section 6.

4. PROGRAMMING

Functional logic programs mostly encode functional computations, but they can also encode non-deterministic computations and computations with incomplete information. This combination simplifies the design of some programs to a degree unparalleled by other programming paradigms. Simpler design leads to simpler proofs of program properties. The focus of this section is on encoding a specification into a program and discussing the correctness of the program with respect to its specification. We keep our discussion informal, and out of necessity, we consider only small examples. In this section, we are not concerned with the efficiency of computations, but we will come back to this issue in Section 6.

We start with an example where the formal specification can be directly written as a program. A *regular expression* over some alphabet is an expression of the following type (we omit empty regular expressions for simplicity):

```
data RE a = Lit a
          | Alt  (RE a) (RE a)
          | Conc (RE a) (RE a)
          | Star (RE a)
```

Except for the syntax, this declaration is identical to the usual definition of regular expression. The type variable `a` ranges over the type of the alphabet, i.e., we do not restrict our alphabet to characters only. A regular expression is one of the following expressions: a letter of the alphabet (`Lit`), a choice between two expressions (`Alt`), a concatenation of two expressions (`Conc`), or zero or more repetitions of an expression (`Star`). For instance, the regular expression ab^* is given a name, `abstar`, and is defined as an expression of type `(RE Char)` as follows:

```
abstar = Conc (Lit 'a') (Star (Lit 'b'))
```

Derived constructors for regular expressions are defined as new operations, e.g.:

```
plus re = Conc re (Star re)
```

The language of a regular expression is defined by a mapping that takes a regular expression and yields a set of words over the given alphabet. We prefer the functional logic programming style and define this mapping as a non-deterministic operation `sem` that yields *any* word (represented as a list) in the language of the given regular expression:

```
sem :: RE a -> [a]
sem (Lit c)    = [c]
sem (Alt  r s) = sem r ? sem s
sem (Conc r s) = sem r ++ sem s
sem (Star r)   = [] ? sem (Conc r (Star r))
```

This is a concise specification of the semantics of regular expressions. Since it is a functional logic program, it is also executable so that we can use it to generate words in the language of a regular expression. For instance, `sem abstar` evaluates to `"a"`, `"ab"`, `"abb"`, ... Therefore, `sem` can be also used to check whether a given word `w` is in the language of a given regular expression `re` through the equation `sem re =:= w`. The correctness of this claim stems from the soundness and completeness of the strategy, which guarantees that the equation is satisfiable if and only if `w` is a value of `sem re`, hence, according to the definition of `sem`, if and only if `w` is in the language of `re`.

Moreover, we can check whether a string `s` contains a word generated by a regular expression `re` (similarly to the Unix's *grep* utility) by solving the equation:

```
xs ++ sem re ++ ys =:= s
    where xs, ys free
```

If `s` contains a word w generated by a regular expression `re` as a substring, w must be a value of `sem re`, and there are sequences `xs` and `ys` such that the concatenation of `xs`, w, and `ys` is identical to `s`. The correctness of our "grep" program again follows from the soundness and completeness of the evaluation strategy. However, there is a noteworthy

difference with the previous case. Earlier, the equation was *verified*, whereas in the case the equation is *solved* for `xs` and `ys`.

If a regular expression r contains the `Star` constructor, the language it generates is infinite so that `sem` r has infinitely many results. Since a demand-driven strategy computes only the values that are required by the context, the previous equation might have a finite search space even in the presence of regular expressions with an infinite language. For instance, the equation

```
xs ++ sem abstar ++ ys =:= "abb"
    where xs, ys free
```

has a finite search space with exactly three solutions in `xs` and `ys`. Observe that we have used `sem` to generate regular expressions satisfying a given property.

For the next example let us consider an informal specification of the *straight selection* sort algorithm: given a non-empty sequence of elements, (1) select the smallest element, (2) sort the remaining ones, and (3) place the selected element in front of them. The only difficult-to-implement step of this algorithm is (1). Step (3) is trivial. Step (2) is obtained by recursion. Step (1) is more difficult because it is specified non-constructively. The specification must define the minimum of a sequence—in the obvious way—but will not say how to compute it. The algorithm to compute the minimum, a problem considerably more complicated than its definition, is left to the programmer. A functional logic programmer, however, can avoid the problem entirely coding a program that "executes" the specification.

The first rule of operation `sort` coded below implements the above specification. The head of the rule employs a functional pattern. The input of `sort` is (evaluated to) a list, and one element of this list, `min`, is non-deterministically selected. The condition of the rule ensures that the selected element is indeed minimal in the list. The remaining code is straightforward. The operation `all`, defined in a standard library (*Prelude*) loaded with every program, returns `True` if its first argument, a predicate, is satisfied by every element of its second argument, a list. The expression `(min <=)` is called a *section*. It it equivalent to a function that takes an argument x and tells whether `min` $\leqslant x$.

```
sort (u++[min]++v)
  | all (min <=) rest
  = min : sort rest
  where rest = u++v
sort [] = []
```

The assertion to establish the correctness of this program states that `sort` l is sorted and contains all and only the elements of l. The proof is by induction on the length of l. The base case is trivial. In the inductive case, if `sort` $l = x : xs$, then the head of the rule ensures that x is an element of l, and the condition of the rule ensures that x is not greater than any element of xs. The induction hypothesis ensures that xs contains all and only the elements of l except x and that xs is sorted. This entails the assertion.

Our final problem is the well-known *missionaries and cannibals puzzle*. Three missionaries and three cannibals want to cross a river with a boat that holds up to two people. Furthermore, the missionaries, if any, on either bank of the river cannot be outnumbered by the cannibals (otherwise, as the intuition hints, they would be eaten). A *state* of the puzzle is represented by the numbers of missionaries and cannibals and the presence of the boat on the *initial* bank of the river:

```
data State = State Int Int Bool
```

Thus, the initial state is "`State 3 3 True`", but we do not construct it directly. Our program constructs a state only through a function, `makeState`, with the same signature as `State`. The function `makeState` applies `State` to its own arguments only if the resulting state is "sensible," otherwise it simply fails. (This programming pattern is called "constrained constructor" in [6].) In this context, a state is sensible only if it is valid according to the numbers of missionaries and cannibals involved in the puzzle and is safe for the missionaries. In the following code, the operators `&&`, `||` and `==` are respectively the Boolean conjunction, disjunction and equality.

```
makeState m c b | valid && safe
                = State m c b
    where valid = 0<=m && m<=3 && 0<=c && c<=3
          safe  = m==3 || m==0 || m==c
```

For example, the initial and final states are constructed by:

```
initial = makeState 3 3 True
final   = makeState 0 0 False
```

There are a total of 10 different moves, five to cross the river in one direction and five in the other direction. Coding all these moves becomes particularly simple with non-determinism and `makeState`. Both contribute to free the code from controlling which move should be executed.

```
move (State m c True)
   = makeState (m-2) c False    - 2 miss
   ? makeState (m-1) c False    - 1 miss
   ? makeState m (c-2) False    - 2 cann
   ? ...
```

A *solution* is a path through the state space from the initial to the final state. A *path* is a sequence of states each of which, except the first one, is obtained from the previous state by a move. The path is represented by a sequence of states. Our program adds new states at the front of a sequence that initially contains only the initial state, hence the order of the states in a path is reversed with respect to the moves that produce a solution. Our program constructs a path only through a function, `extendPath`, that takes a non-empty path `p` and adds a state at the front of `p` so that the resulting path is sensible. In this context, a path is sensible only if it does not contain a cycle and each state of the path except the initial one is obtained with a move from the preceding state.

```
extendPath p | noCycle = next : p
    where next    = move (head p)
          noCycle = all (next /=) p
```

Computing a solution of the puzzle becomes particularly simple with non-determinism and `extendPath`. Our program simply extends a path until it produces the final state.

```
main = extendToFinal [initial]

extendToFinal p =
    if (head p == final) then p
    else extendToFinal (extendPath p)
```

The assertion to establish the correctness of this program states that operation `main` terminates with a solution of the

puzzle if and only if the puzzle has a solution. We discuss the properties of the program that inspire confidence in its correctness and would be key to a more rigorous proof.

The program maintains two invariants: (1) Every state constructed during an execution does not violate the problem's constraints, and (2) the value returned by every invocation of operation `extentToFinal` is a path in the state space. Invariant (1) stems from the fact that a state is constructed only by operation `makeState`, which ensures the legality of a state through conditions `valid` and `safe`. Invariant (2) stems from invariant (1) and the fact that a path is constructed only by operation `extendPath`, which ensures that the sequence of states it returns is indeed a path since only states originating from valid moves are added to a path. Operation `extendToFinal` keeps extending its argument, a path, unless the last inserted state is the goal, in which case it returns its argument. Operation `extendToFinal` is invoked by `main` with the initial state. Hence, if operation `extendToFinal` successfully terminates, it returns a solution of the problem. Operation `extendPath` invokes operation `move`, which is non-deterministic. The completeness of the evaluation strategy ensures that every move available in a state is chosen. Hence, if the problem has a solution, `extendToFinal`, and consequently `main`, will return this solution.

A further property of the program is that any solution returned by `extendToFinal` contains no repeated states. This condition stems from the fact that the paths returned by `extendToFinal` are constructed by operation `extendPath` which, through condition `noCycle`, fails to insert a state in a path that already contains that state.

5. LOOKING BEHIND

A motivation behind the emergence of functional logic programming was the desire to unify the two most prominent branches of the declarative paradigms, functional and logic programming. Much progress has been made toward this goal, and in the process the goal has somewhat changed. Unification in the form of a single common language accepted and used by the two communities, as perhaps some of us may have envisioned, does not appear near. It may be difficult to understand all the reasons of this development, but investments, personal preferences, and specialization are certainly contributing factors. However, each community has become much more aware of the other, there are conferences in which papers from each paradigm are understood and equally enjoyed by all the participants, and efforts such as introducing logic variables in functional languages [15] and functions in logic languages [14] are indisputable evidence that the gap between the two paradigms has been narrowed. More interestingly, functional logic programming has become a paradigm in its own right with results whose depth and solidity rival those of the other declarative paradigms.

We sketched the concepts of Curry and used it for concrete examples. Curry is currently the only functional logic language which is based on strong theoretical foundations (e.g., sound, complete, and optimal evaluation strategies [4, 5]) and has been used for a variety of projects, such as web-based information systems, embedded systems programming and e-learning systems [24, 25]. Curry has been also used from the very beginning to teach functional and logic programming concepts with a single programming language [20]. Nevertheless, there are various other languages with similar goals. We briefly discuss some of them and relate them to Curry.

The language $\mathcal{TOY}$ [30] has strong connections to Curry since it is based on similar foundations. In contrast to Curry, it supports more advanced constraint structures (e.g., disequality constraints) but misses application-oriented libraries so that it has not been used for application programming. This is also the case for Escher [29], a functional logic language where functions are deterministically evaluated in a demand-driven manner and logic features are modelled by simplification rules for logical expressions.

The language Oz [38] is based on a computation model that extends concurrent constraint programming with features for distributed programming and stateful computations. It supports functional notation, but operations used for goal solving must be defined with explicit disjunctions. Thus, functions used to solve equations must be defined differently from functions to rewrite expressions.

Since functions can be considered as specific relations, there are also approaches to extend logic languages with features for functional programming by adding syntactic sugar for functional notation. For instance, [14] proposes to add functional notation to Ciao-Prolog which is translated by a preprocessor into Prolog. The functional logic language Mercury [39] restricts logic programming features in order to provide a highly efficient implementation. In particular, operations must have distinct modes so that their arguments are either completely known or unbound at call time. This inhibits the application of typical logic programming techniques, e.g., computing with structures that are only partially known, so that some programming techniques for functional logic programming [6, 26] cannot be applied with Mercury. This condition has been relaxed in the language HAL [18] which adds constraint solving possibilities. However, both languages are based on a strict operational semantics that does not support optimal evaluation as in functional programming.

Some of the concepts and techniques developed for functional logic programming have migrated to other areas of computer science. A success story is Tim Sheard's Ωmega system [36]. Ωmega is a type systems with an infinite hierarchy of computational levels that combines programming and reasoning. At the value level computation is performed by reduction. At all higher levels computation is performed by narrowing in the inductively sequential [2] systems using definitional trees to execute only needed steps.

6. LOOKING AHEAD

Will functional logic programs ever execute as efficiently as imperative programs? We do not have the final answer, but the future is promising. Needed narrowing, the strategy for a core functional logic language, exhibits two distinct aspects of non-determinism: *don't care* and *don't know* choices. Don't care choices occur in some function calls when more than one subexpression needs to be evaluated, such as when both arguments of an equation or of an arithmetic binary operation are demanded. Because the language is free of side effects, we do not care which argument is evaluated first. Therefore, it is viable to evaluate both arguments concurrently. A small amount of synchronization may be necessary to determine when the evaluation of a set of don't care choices is completed and to instantiate variables shared by these choices.

Don't know choices occur in some function calls when more than one alternative is available, such as when a free variable argument is instantiated with different values and/or when an expression is reduced by different rules. Since we don't know in advance which alternative(s) will produce a solution (if we did, we would have coded a program that selects these alternatives only), all the alternatives must be sampled to ensure the completeness of the computation. Because each alternative is independent of any other alternative, in this situation too, the order of execution of the alternatives is irrelevant. As a consequence, both don't care and don't know choices can be executed concurrently. This is one area where the abstraction from the evaluation order pays off. For many problems, in particular non-numeric ones, functional logic programs have a degree of parallelism that is potentially higher than corresponding imperative programs. Future research will investigate this potential and attempt to exploit it for faster execution of functional logic programs.

The most important aids to the development of real applications, after a solid and efficient compiler, are a set of libraries for application programming and environments and tools that support program development and maintenance. Promising work has been started in both areas. Functional logic programming supports abstractions that lead to high-level, declarative programming interfaces. Libraries with interfaces with these characteristics ease the construction of reliable applications in various application areas, as shown by the various libraries developed for Curry (see Section 2). The formal foundations of functional logic languages are also useful for the development of environments and tools for reasoning about programs. Developments in these areas include tools for program analysis [23], verification [16], partial evaluation [1], profiling [11], debugging [10], and testing [17].

These initial efforts demonstrate the suitability of functional logic programming techniques for application programming. Some applications have been successfully developed, as already mentioned above. Due to the availability of Curry implementations with support for meta-programming, most of the current tools are implemented in Curry. These applications demonstrate the potential of functional logic programming. Future research and development in these areas will promote and ease the use of functional logic programs in practice.

The major asset of functional logic programming is the amalgamation of functional and logic programming features. Logic programming emphasizes non-determinism, whereas functional programming is deterministic, that is, the input of a computation completely determines the output. This dichotomy creates a conflict when it is desirable to reason "functionally" about non-deterministic computations, for example, to select a minimal or maximal value among a set of results of a non-deterministic computation, or to print all these results in some order. Curry provides some features to encapsulate non-deterministic computations. For example, the set of results of a non-deterministic computation can be collected in some data structure and processed in its entirety. These features are useful and are used in some applications [13, 31] but have limitations. Future research on the theory of these features and on their implementation should ease the encoding of complex problems into relatively simple functional logic programs.

7. CONCLUSION

Declarative languages describe programming tasks through high-level, concise and executable abstractions. Other paradigms offer similar abstractions, but to a lesser degree. Compared to purely functional languages, functional logic languages allow non-deterministic descriptions with partial information that simplify encoding some problems into programs. Compared to purely logic languages, functional logic languages allow functional composition, which improves code modularity, reuse and efficiency of execution.

Functional logic programs have the flavor of executable specifications or high-level descriptions. Correctness proofs for these programs are simpler than those in other programming paradigms. These executable specifications are a good solution of some programming problems, for example, when the execution is efficient or when no other algorithm is known. Of course, the performance of this specification-oriented approach may not be satisfactory for some problems. Even in this situation, the effort to develop an initial prototype is not wasted. First, building one or more prototypes is a reasonable step for many software artifacts since a prototype clarifies problems and allows experimentation at a reduced cost. Second, a prototype can be refined with the introduction of more efficient data structures (e.g., replace a list with a tree) or more efficient algorithms (e.g., replace linear search with binary search). Since declarative languages also allow the implementation of efficient data structures [33], the stepwise improvement of a prototype can be done without changing the implementation language. This methodology has the advantage that the initial prototype serves as a specification for verifying the improved program and/or as an oracle for testing it.

Last but not least, programming in a functional logic language is fun. As we move through the evolution of programming languages sketched in the introduction, we find that programming problems become easier and solutions more reliable. Evaluating an arithmetic expression directly through its standard notation without using registers, temporaries and stacks is safer and more convenient. Structuring programs without labels and *gotos* is more elegant and increases the confidence in the correctness of the result. Legitimate concerns about the efficiency of these revolutionary innovations were raised at the times of their introductions. History repeats itself. Programming with non-determinism is exhilarating when a potentially difficult task is replaced by an almost effortless choice followed by a much simpler constraint. In our examples, this was to determine whether a string belongs to the language defined by a regular expression, or to find the minimum of a sequence, or to produce the solution of a puzzle without any concern for selecting the moves. This approach is not always the best one in practice. For example, the non-deterministic choice of the minimum of a list may have to be replaced by an algorithm that is much faster for the computer to execute and much slower for the programmer to code and verify. But also in these cases, there is understanding and value in the prototypical implementation and satisfaction in its simplicity and elegance.

Acknowledgments.

This work was partially supported by the German Research Council (DFG) grant Ha 2457/5-2, the DAAD grants D/06/29439 and D/08/11852, and the NSF grants CCR-0110496 and CCR-0218224.

8. REFERENCES

[1] E. Albert, M. Hanus, and G. Vidal. A practical partial evaluator for a multi-paradigm declarative language. *Journal of Functional and Logic Programming*, 2002(1), 2002.

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

[3] S. Antoy. Constructor-based conditional narrowing. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, pages 199–206. ACM Press, 2001.

[4] S. Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 40(1):875–903, 2005.

[5] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.

[6] S. Antoy and M. Hanus. Functional logic design patterns. In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, pages 67–87. Springer LNCS 2441, 2002.

[7] S. Antoy and M. Hanus. Declarative programming with function patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pages 6–22. Springer LNCS 3901, 2005.

[8] S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*, pages 87–101. Springer LNCS 4079, 2006.

[9] M. Bezem, J. W. Klop, and R. de Vrijer (eds.). *Term Rewriting Systems*. Cambridge University Press, 2003.

[10] B. Brassel, S. Fischer, M. Hanus, F. Huch, and G. Vidal. Lazy call-by-value evaluation. In *Proc. of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*, pages 265–276. ACM Press, 2007.

[11] B. Braßel, M. Hanus, F. Huch, J. Silva, and G. Vidal. Run-time profiling of functional logic programs. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, pages 182–197. Springer LNCS 3573, 2005.

[12] B. Braßel, M. Hanus, and M. Müller. High-level database programming in Curry. In *Proc. of the Tenth International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, pages 316–332. Springer LNCS 4902, 2008.

[13] B. Braßel and F. Huch. On a tighter integration of functional and logic programming. In *Proc. APLAS 2007*, pages 122–138. Springer LNCS 4807, 2007.

[14] A. Casas, D. Cabeza, and M.V. Hermenegildo. A syntactic approach to combining functional notation, lazy evaluation, and higher-order in LP systems. In *Proc. of the 8th International Symposium on Functional and Logic Programming (FLOPS 2006)*, pages 146–162. Springer LNCS 3945, 2006.

[15] K. Claessen and P. Ljunglöf. Typed logical variables in Haskell. In *Proc. ACM SIGPLAN Haskell Workshop*, Montreal, 2000.

[16] J.M. Cleva, J. Leach, and F.J. López-Fraguas. A logic programming approach to the verification of functional-logic programs. In *Proceedings of the 6th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 9–19. ACM Press, 2004.

[17] S. Fischer and H. Kuchen. Systematic generation of glass-box test cases for functional logic programs. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07)*, pages 75–89. ACM Press, 2007.

[18] M.J. García de la Banda, B. Demoen, K. Marriott, and P.J. Stuckey. To the gates of HAL: A HAL tutorial. In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, pages 47–66. Springer LNCS 2441, 2002.

[19] J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.

[20] 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.

[21] M. Hanus. Type-oriented construction of web user interfaces. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'06)*, pages 27–38. ACM Press, 2006.

[22] M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.

[23] M. Hanus. Call pattern analysis for functional logic programs. In *Proceedings of the 10th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'08)*, pages 67–78. ACM Press, 2008.

[24] M. Hanus and K. Höppner. Programming autonomous robots in Curry. *Electronic Notes in Theoretical Computer Science*, 76, 2002.

[25] M. Hanus and F. Huch. An open system to support web-based learning. In *Proc. 12th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2003)*, pages 269–282, 2003.

[26] M. Hanus and C. Kluß. Declarative programming of user interfaces. In *Proc. of the 11th International Symposium on Practical Aspects of Declarative Languages (PADL'09)*, pages 16–30. Springer LNCS 5418, 2009.

[27] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at `http://www.curry-language.org`, 2006.

[28] H. Hussmann. Nondeterministic algebraic specifications and nonconfluent term rewriting. *Journal of Logic Programming*, 12:237–255, 1992.

[29] J. Lloyd. Programming in an integrated functional

and logic language. *Journal of Functional and Logic Programming*, (3):1–49, 1999.

[30] F. López-Fraguas and J. Sánchez-Hernández. TOY: A multiparadigm declarative system. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.

[31] W. Lux. Implementing encapsulated search for a lazy functional logic language. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pages 100–113. Springer LNCS 1722, 1999.

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

[33] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[34] S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.

[35] Uday S. Reddy. Narrowing as the operational semantics of functional languages. In *Proceedings of the IEEE International Symposium on Logic in Computer Science*, pages 138–151, Boston, 1985.

[36] T. Sheard. Type-level computation using narrowing in Ωmega. *Electron. Notes Theor. Comput. Sci.*, 174(7):105–128, 2007.

[37] J.R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of the ACM*, 21(4):622–642, 1974.

[38] 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.

[39] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.