Logic Based Program Synthesis and Transformation: 15th Int'nl Symp., LOPSTR 2005
London, UK, September 7–9, 2005


# Declarative Programming with Function Patterns*

Sergio Antoy[1] Michael Hanus[2]

[1] Computer Science Dept., Portland State University, Oregon, U.S.A.
antoy@cs.pdx.edu

[2] Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany.
mh@informatik.uni-kiel.de

**Abstract.** We propose an extension of functional logic languages that allows the definition of operations with patterns containing other defined operation symbols. Such "function patterns" have many advantages over traditional constructor patterns. They allow a direct representation of specifications as declarative programs, provide better abstractions of patterns as first-class objects, and support the high-level programming of queries and transformation of complex structures. Moreover, they avoid known problems that occur in traditional programs using strict equality. We define their semantics via a transformation into standard functional logic programs. Since this transformation might introduce an infinite number of rules, we suggest an implementation that can be easily integrated with existing functional logic programming systems.

## 1 Motivation

Functional logic languages (see [16] for a survey) integrate the most important features of functional and logic languages to provide a variety of programming concepts to the programmer. For instance, the concepts of demand-driven evaluation, higher-order functions, and polymorphic typing from functional programming are combined with logic programming features like computing with partial information (logic variables), constraint solving, and non-deterministic search for solutions. This combination, supported by optimal evaluation strategies [6] and new design patterns [8], leads to better abstractions in application programs such as implementing graphical user interfaces [18] or programming dynamic web pages [19].

A functional logic program consists of a set of datatype definitions and a set of functions or operations, defined by equations or rules, that operate on these types. For instance, the concatenation operation "`++`" on lists can be defined by the following two rules, where "`[]`" denotes the empty list and "`x:xs`" the non-empty list with first element `x` and tail `xs`:

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

Expressions are evaluated by rewriting with rules of this kind. For instance, `[1,2]++[3]` evaluates to `[1,2,3]`, where $[x_1,x_2,\ldots,x_n]$ denotes $x_1 : x_2 : \ldots : x_n : []$, in three rewrite steps:

* This work was partially supported by the German Research Council (DFG) under grant Ha 2457/5-1 and the NSF under grant CCR-0218224.

```
[1,2]++[3]  →  1:([2]++[3])  →  1:(2:([]++[3]))  →  [1,2,3]
```

Beyond such functional-like evaluations, functional logic languages also compute with unknowns (logic variables). For instance, a functional logic language is able to *solve* an equation like `xs++[x] =:= [1,2,3]` (where `xs` and `x` are logic variables) by guessing the bindings `[1,2]` and `3` for `xs` and `x`, respectively.

This constraint solving capability can be exploited to define new operations using already defined functions. For instance, the operation `last`, which yields the last element of a list, can be defined as follows (the "`where...free`" clause declares logic variables in rules):

```
last l | xs++[x] =:= l = x   where xs,x free          (last1)
```

In general, a *conditional equation* has the form $l$ `|` $c$ `=` $r$ and is applicable for rewriting if its condition $c$ has been solved. A subtle point is the meaning of the symbol "`=:=`" used to denote equational constraints. Since modern functional logic languages, like Curry [17, 22] or Toy [25], are based on a non-strict semantics [6, 14] that supports lazy evaluation and infinite structures, it is challenging to compare arbitrary, in particular infinite, objects. Thus, the equality symbol "`=:=`" in a condition is usually interpreted as *strict equality*—the equation $t_1$ `=:=` $t_2$ is satisfied iff $t_1$ and $t_2$ are reducible to the same *constructor term* (see [13] for a more detailed discussion on this topic). A constructor term is a fully evaluated expression; a formal definition appears in Section 3.

Strict equality evaluates both its operands to a constructor term to prove the validity of the condition. For this reason, the strict equation "`x =:= head []`" does not hold for any `x`. The operation `head` is defined by the single rule `head (x:xs) = x`. Therefore, the evaluation of `head []` fails to obtain a constructor term. While the behavior of "`=:=`" is natural and intuitive in this example, it is less so in the following example.

A consequence of the strict equality in the definition of `last` in Display (*last1*) is that the list argument of `last` is fully evaluated. In particular, `last [failed,2]`, where `failed` is an operation whose evaluation fails, has no result. This outcome is unnatural and counterintuitive. In fact, the usual functional recursive definition of `last` would produce the expected result, 2, for the same argument. Thus, strict equality is harmful in this example (further examples will be shown later) since it evaluates more than one intuitively requires and, thus, reduces the inherent laziness of the computation.

There are good reasons for the usual definition of strict equality [13]; we will see that just dropping the strictness requirements in equational conditions leads to a non-intuitive behavior. Therefore, we propose in this paper an extension of functional logic languages with a new concept that solves all these problems: *function patterns*. Traditional patterns (i.e., the arguments of the left-hand sides of rules) are required to be constructor terms. Function patterns can also contain defined operation symbols so that the operation `last` is simply defined as

```
last (xs++[x]) = x
```

This definition leads not only to concise specifications, but also to a "lazier" behavior. Since the pattern variables `xs` and `x` are matched against the actual (possibly unevaluated) parameters, with this new definition of `last`, the expression `last [failed,2]` evaluates to 2.

The next section defines the notations used in this paper. Section 3 defines the concept of function patterns, and Section 4 shows examples of its use. Section 5 proposes an

implementation of function patterns and shows its performance on some benchmarks. We compare our approach with related work in Section 6 and conclude in Section 7.

## 2 Preliminaries

In this section we review some notations for term rewriting [9, 10] and functional logic programming [16] concepts used in the remaining of this paper.

Since polymorphic types are not important for our proposal, we ignore them and consider a many-sorted *signature* $\Sigma$ partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{F}$ of (defined) *functions* or *operations*. We write $c/n \in \mathcal{C}$ and $f/n \in \mathcal{F}$ for $n$-ary constructor and operation symbols, respectively. Given a set of variables $\mathcal{X}$, the set of *terms* and *constructor terms* are denoted by $\mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ and $\mathcal{T}(\mathcal{C}, \mathcal{X})$, respectively. As in the concrete syntax of Curry, we indicate the application of a function to arguments by juxtaposition, e.g., $f\ t_1 \ldots t_n$. A term is *linear* if it does not contain multiple occurrences of a variable. A term is *operation-rooted* (*constructor-rooted*) if its root symbol is an operation (constructor). A *head normal form* is a term that is not operation-rooted, i.e., it is a variable or a constructor-rooted term.

Given a signature $\Sigma = (\mathcal{C}, \mathcal{F})$, a *functional logic program* is set of rules of the form

$$f\ d_1 \ldots d_n \mid c\ =\ e$$

where $f/n \in \mathcal{F}$ is the function defined by this rule, $d_1, \ldots, d_n$ are constructor terms (also called *patterns*) such that the *left-hand side* $f\ d_1 \ldots d_n$ is linear, the *condition* $c$ (which can be omitted) is a constraint, and the *right-hand side* is an expression.[3] A *constraint* is any expression of type `Success`, e.g., the trivial constraint `success` which is always satisfied, the equational constraint $t_1$ `=:=` $t_2$ which is satisfied if both sides are reducible to the same constructor term, or the conjunction $c_1$ `&` $c_2$ which is satisfied if both arguments are satisfied (operationally, "`&`" is the basic concurrency combinator since both arguments are evaluated concurrently). To provide a simple operational meaning of conditional rules, we consider the rule "$l \mid c\ =\ r$" as equivalent to the unconditional rule "$l\ =$ `cond` $c\ r$" where the auxiliary operation `cond` is defined by

```
cond success x = x
```

Note that rules can overlap so that operations can be *non-deterministic*. For instance, the rules

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

define a non-deterministic operation "`?`" that returns one of its arguments, and

```
insert x []     = [x]
insert x (y:ys) = x : y : ys  ?  y : insert x ys
```

define a non-deterministic insertion of an element into a list.

We need a few additional notions to formally define computations w.r.t. a given program. A *position* $p$ in a term $t$ is represented by a sequence of natural numbers.

[3] In the concrete syntax, the variables $x_1, \ldots, x_k$ occurring in $c$ or $e$ but not in the left-hand side must be explicitly declared by a where-clause (`where` $x_1, \ldots, x_k$ `free`) to enable some consistency checks.

Positions are used to identify specific subterms. Thus, $t|_p$ denotes the *subterm* of $t$ at position $p$, and $t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term $s$ (see [10] for details). A *substitution* is an idempotent mapping $\sigma : \mathcal{X} \to \mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$. Substitutions are extended to morphisms on terms in the obvious way. A *rewrite step* $t \to t'$ is defined w.r.t. a given program $P$ if there are a position $p$ in $t$, a rule $l = r \in P$ with fresh variables, and a substitution $\sigma$ with $t|_p = \sigma(l)$ and $t' = t[\sigma(r)]_p$. A term $t$ is called *irreducible* or in *normal form* if there is no term $s$ such that $t \to s$.

Functional logic languages also compute solutions of free variables occurring in expressions by instantiating them to constructor terms so that a rewrite step becomes applicable. The combination of variable instantiation and rewriting is called *narrowing*. Formally, $t \leadsto_\sigma t'$ is a *narrowing step* if $\sigma(t) \to t'$ and $t|_p$ is not a variable for the position $p$ used in this rewrite step. Although the latter condition is a substantial restriction on the possible narrowing steps, there are still too many possibilities to apply this definition of narrowing in practice. Therefore, older narrowing strategies (see [16] for a detailed account), influenced by the resolution principle, require that the substitution used in a narrowing step is a most general unifier of $t|_p$ and the left-hand side of the applied rule. As shown in [6], this condition prevents the development of optimal evaluation strategies. Therefore, many recent narrowing strategies relax this requirement but provide other constructive methods to compute a small set of unifiers and positions used in narrowing steps [5]. In particular, *needed* or *demand-driven* strategies perform narrowing steps only if they are necessary to compute a result. For instance, consider the following program containing a declaration of natural numbers in Peano's notation and operations for addition and a "less than or equal" test (the pattern "`_`" denotes an unnamed *anonymous variable*):

```
data Nat = O | S Nat

add O     y = y                  leq O     _     = True
add (S x) y = S (add x y)        leq (S _) O     = False
                                 leq (S x) (S y) = leq x y
```

Then the subterm $t$ in the expression "`leq O` $t$" need not be evaluated since the first rule for `leq` is directly applicable. On the other hand, the first argument of "`leq (add v w) O`" must be evaluated to rewrite this expression. Furthermore, the expression "`leq v (S O)`" becomes reducible after the instantiation of `v` to either `O` or `S z`.

This strategy, called *needed narrowing* [6], is optimal for the class of inductively sequential programs that do not allow overlapping left-hand sides. Its extension to more general programs with possibly overlapping left-hand sides can be found in [3, 4]. A precise description of this strategy with the inclusion of sharing, concurrency, and external functions is provided in [1]. In the following, we denote by $t \overset{*}{\leadsto}_\sigma t'$ a sequence of needed narrowing steps evaluating $t$ to $t'$, where $\sigma$ is the composition of all the substitutions applied in the sequence's steps.

## 3 Function Patterns

As already mentioned, strict equality is the usual interpretation of equational conditions in functional logic languages based on a non-strict semantics. Since this looks like a

contradiction, first we explain the reasons for using strict equality, then we explain our proposal.

Strict equality holds between two expressions if both can be reduced to a same constructor term. Consequently, strict equality is not reflexive; e.g., "`head [] =:= head []`" does not hold. One motivation for this restriction is the difficulty of solving equations with a reflexive meaning in the presence of non-terminating computations. For instance, consider the following functions:

```
from x = x : from (x+1)

rtail (x:xs) = rtail xs
```

Demanding reflexivity in equational conditions implies that the condition

```
rtail (from 0) =:= rtail (from 5)
```

should hold since both sides are reducible to `rtail (from` $n$`)` for every $n \geq 5$. However, this is not a normal form, so it is unclear how far some side of the equation should be reduced. Although this problem could be solved by an exhaustive search of the infinite reduction space (clearly not a practical approach), there are also problems with reflexivity in the presence of infinite data structures. For instance, the condition

```
from 0 =:= from 0
```

should hold if "`=:=`" is reflexive. Since both sides describe the infinite list of natural numbers, the condition

```
from 0 =:= from2 0
```

should also hold w.r.t. the definition

```
from2 x = x : x+1 : from2 (x+2)
```

Obviously, the equality of infinite structures defined by syntactically different functions is undecidable in general (since this requires solving the halting problem). Therefore, one needs to restrict the meaning of equational conditions to a non-reflexive interpretation. Note that this condition is not specific to functional logic languages: Haskell [27] also defines the equality symbol "`==`" as strict equality by default (this could be changed by the use of type classes).

Although the previous examples have shown that there are good reasons to avoid reflexivity of equality, one might think that the evaluation of some parts of an expression in an equational condition is unnecessary or even unintended, as discussed with the function `last` defined in Section 1. For instance, one could propose to relax strict equality as follows: to solve the equation $x$ `=:=` $t$, bind $x$ to $t$ (instead of binding $x$ to the evaluation of $t$). Although in some cases, such as the operation `last`, this policy seems to produce the desired behavior, in other cases it would lead to a non-intuitive behavior. For instance, consider the function:

```
f x | x =:= from 0 = 99
```

The expression "`let x free in f x`" would evaluate to `99` since `x` would be bound to `from 0` which would not be further evaluated. Similarly, "`let x free in (f x,99)`" would evaluate to `(99,99)`. However, the evaluation of "`let x free in (f x,f x)`" would not terminate, since the evaluation of the equational condition "`from 0 =:= from 0`" would not terminate. In fact, "`f x`"

should be evaluated twice, the first time with `x` unbound and the second time with `x` bound to "`from 0`".

This example shows that a simple binding of logic variables to unevaluated expressions is also problematic. Therefore, non-strict functional logic languages usually bind logic variables only to constructor terms. However, *pattern variables*, i.e., variables occurring in patterns, can be bound to unevaluated expressions. Thus, in order to relax the strictness conditions in equational conditions, one needs a finer control over the kind of involved variables, i.e., one needs to distinguish between logic variables that are bound to constructor terms and pattern variables that can be bound to unevaluated expressions. For this purpose, we propose to use function patterns as an intuitive and simple solution to the problems discussed above. We explain the details below.

A *function pattern* is a pattern that contains, in addition to pattern variables and constructor symbols, defined operation symbols. For instance, if "`++`" is the list concatenation operation defined in Section 1, `(xs++[x])` is a function pattern. Using function patterns, we can define the function `last` as

```
last (xs++[x]) = x                                    (last2)
```

This definition not only is concise but also introduces `xs` and `x` as pattern variables rather than logic variables as in definition (*last1*) above. Since pattern variables can be bound to unevaluated expressions, `last` returns the last element of the list without evaluating any element of the list. For instance, `last [failed,2]` evaluates to `2`, as intended.

To extend functional logic languages with function patterns, we have to clarify two issues: what is the precise meaning of function patterns, and how are operations defined using function patterns executed? We prefer to avoid the development of a new theory of such extended functional logic programs and to reuse existing results about semantics and models of traditional functional logic programs (e.g., [14]). Therefore, we define the meaning of such programs by a transformation into standard programs. The basic idea is to transform a rule containing a function pattern into a set of rules where the function pattern is replaced by its evaluation(s) to a constructor term.

Consider the definition (*last2*) above. The evaluations of `xs++[x]` to a constructor term are

$$\begin{array}{ll} \texttt{xs++[x]} \overset{*}{\leadsto}_{\texttt{xs}\mapsto\texttt{[]}} & \texttt{[x]} \\ \texttt{xs++[x]} \overset{*}{\leadsto}_{\texttt{xs}\mapsto\texttt{[x1]}} & \texttt{[x1,x]} \\ \texttt{xs++[x]} \overset{*}{\leadsto}_{\texttt{xs}\mapsto\texttt{[x1,x2]}} & \texttt{[x1,x2,x]} \\ \ldots & \end{array}$$

Thus, the single rule (*last2*) is an abbreviation of the set of rules

```
last [x] = x
last [x1,x] = x
last [x1,x2,x] = x
...
```

These rules exactly describe the intended meaning of the operation `last`. Obviously, this transformation cannot be done at compile time since it may lead to an infinite set of program rules. Therefore, in Section 5 we discuss techniques to perform this transformation at run time. The basic idea is, for an invocation of `last` with argument

$l$, to compute the single ordinary (constructor) pattern $p$ of the rule that would be fired by that invocation and to match, or more precisely to unify since we narrow, $l$ and $p$.

This idea has two potential problems. First, we have to avoid its potential underlying circularity. Evaluating a function pattern must not involve executing the operation being defined since its definition is not available before the function pattern has been evaluated. For instance, a rule like

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

is not allowed since the meaning of the function pattern `(xs++ys)` depends on the definition of "`++`". In order to formalize such dependencies, we introduce level mappings.

**Definition 1.** A *level mapping* $l$ for a functional logic program $P$ is a mapping from functions defined in $P$ to natural numbers such that, for all rules $f\ t_1 \ldots t_n \mid c = e$, if $g$ is a function occurring in $c$ or $e$, then $l(g) \leq l(f)$. □

For instance, consider the program $P$ consisting of the rule (*last1*) and the rules defining "`++`". Then $l(\texttt{++}) = 0$ and $l(\texttt{last}) = 1$ is a possible level mapping for $P$. Using level mappings, we can define the class of acceptable programs.

**Definition 2.** A functional logic program $P$ with function patterns is *stratified* if there exists a level mapping $l$ for $P$ such that, for all rules $f\ t_1 \ldots t_n \mid c = e$, if $g$ is a defined function occurring in some $t_i$ ($i \in \{1, \ldots, n\}$), then $l(g) < l(f)$. □

The restriction to stratified programs ensures that, if an operation $f$ is defined using a function pattern $p$, the evaluation of $p$ to a constructor term does not depend, directly or indirectly, on $f$.

The second potential problem of our intended transformation of rules containing function patterns is nonlinearity. For instance, consider the operation `idpair` defined by

```
idpair x = (x,x)
```

and the rule

```
f (idpair x) = 0                                        (f1)
```

After evaluating the function pattern `(idpair x)`, the rule (*f1*) would be transformed into

```
f (x,x) = 0                                             (f2)
```

However, this rule is not left-linear and therefore is not allowed in traditional functional logic programs. Relaxing the left-linearity condition on rules is not viable since it causes difficulties similar to those we discussed for relaxing strict equality. Usually, the intended meaning of multiple occurrences of a variable in the left-hand side is that the actual arguments at these variables' positions should be equal in the sense of an equational condition [4]. This can be expressed by introducing new pattern variables and equational conditions. Thus, the rule (*f2*) is finally transformed into the valid rule

```
f (x,y) | x=:=y = 0
```

The above considerations motivate the following interpretation of functional logic programs with function patterns.

**Definition 3.** Let $P$ be a stratified functional logic program with function patterns. The meaning of $P$ is the program $P^*$ defined by:

$$P^* = \{lin(f\ t_1 \ldots t_n \mid \sigma(c) = \sigma(e)) \quad \text{s.t. } f\ e_1 \ldots e_n \mid c = e \in P, \\ (e_1, \ldots, e_n) \overset{*}{\leadsto}_\sigma (t_1, \ldots, t_n), \text{ and} \\ t_1, \ldots, t_n \text{ are constructor terms}\}$$

where $lin$ denotes the linearization of a rule defined by

$$lin(l \mid c = r) = \begin{cases} l \mid c = r & \text{if } l \text{ is linear;} \\ lin(l[y]_q \mid (x \texttt{=:=} y \,\&\, c) = r) & \text{if } l|_p = x = l|_q,\ p \neq q, \\ & x \text{ is a variable and } y \text{ is fresh.} \end{cases}$$ □

The associated program $P^*$ is well defined (since $P$ is stratified) and a valid functional logic program, since the patterns in the left-hand sides are linear constructor terms. Since function patterns are transformed into ordinary patterns, the variables occurring in function patterns become ordinary pattern variables that can be bound to unevaluated expressions. Thus, function patterns relax the strict evaluation conditions of strict equality without any difficulties. The only potential problem is the generation of an infinite number of rules for $P^*$ in case of function patterns involving recursive functions. Therefore, we show in Section 5 a transformation, executed at run time, that generates only the rules that are required for a specific application of an operation defined by a function pattern.

## 4 Examples

In this section we present a few more examples of programs that use function patterns. The following example makes essential use of function patterns. The proposed design would not work with strict equality.

*Example 1.* This example is a problem of the 1993 East-Central Regionals of the ACM International Collegiate Programming Contest. Given a number $n$, we form the chain of $n$ by:

(1) arranging the digits of $n$ in descending order,
(2) arranging the digits of $n$ in ascending order,
(3) subtracting the number obtained in (2) from the number obtained in (1) to form a new number, and
(4) repeating these steps for the new number.

E.g., the chain of 123 is 198, 792, 693, 594, 495, 495... The problem is to compute the length of the chain up to the first repeated number—seven for 123.

The implementation is simpler if one separates the task of constructing the chain of a number from the task of finding the first repeated element in the chain. The solution of the problem is obtained by ("`.`" denotes function composition):

```
lengthUpToRepeat . chain
```

where the function `chain` constructs the chain of a number and the function `lengthUpToRepeat` measures the length of the chain up to the first repeated element. The latter function, coded below, uses a function pattern.

```
lengthUpToRepeat (p++[r]++q)
    | nub p == p && elem r p
    = length p + 1
```

The pattern breaks the infinite chain of a number into a prefix `p`, the first repeated element `r`, and the rest of the chain `q`. The symbols `nub` and `elem` denote library functions. `nub` removes repeated elements from a list. Hence, the condition `nub p == p` ensures that there are no repeated elements in `p`. `elem` tells whether an element is a member of list. The conjunction of the two conditions ensures that `r` is the first repeated element in the chain.

By contrast, an implementation that uses strict equality, i.e., that attempts to solve `p++[r]++q =:= chain n`, would be flawed. By design, `chain n` is an infinite list, and therefore strict equality would not terminate. □

The second example shows the use of function patterns to specify transformations in tree-like structures.

*Example 2.* This example addresses the simplification of symbolic arithmetic expressions. E.g., $1 * (x + 0)$ simplifies to $x$. We define expressions as

```
data Exp = Lit Int | Var [Char] | Add Exp Exp | Mul Exp Exp
```

The following non-deterministic function, `evalTo`, defines a handful of expressions that for every expression $e$ evaluate to $e$ itself. Obviously, many more are possible, but the following ones suffice to make our point.

```
evalTo e = Add (Lit 0) e
         ? Add e (Lit 0)
         ? Mul (Lit 1) e
         ? Mul e (Lit 1)
```

The following function replaces in an expression a subexpression identified by a position with another subexpression.

```
replace _ [] x = x
replace (Add l r) (1:p) x = Add (replace l p x) r
replace (Add l r) (2:p) x = Add l (replace r p x)
replace (Mul l r) (1:p) x = Mul (replace l p x) r
replace (Mul l r) (2:p) x = Mul l (replace r p x)
```

Observe that `replace` $c$ $p$ $e$, where $c$ is a "context", $p$ is a position and $e$ is an expression, is the term replacement operation denoted by $c[e]_p$ in Section 2. Finally, the simplification operation, `simplify`, replaces in a context `c` an expression that evaluates to `x` with `x` itself. `p` is the position of the replacement in the context.

```
simplify (replace c p (evalTo x)) = replace c p x
```

E.g., if $t_1 =$ `Mul (Lit 1) (Add (Var "x") (Lit 0))`, then `simplify` $t_1$ evaluates to $t_2 =$ `Add (Var "x") (Lit 0)` and `simplify` $t_2$ evaluates to `Var "x"`. If an expression $t$ cannot be simplified, `simplify` $t$ fails; otherwise, it non-deterministically executes a single simplification step. The application of repeated simplification steps to an ex-

pression until no more simplification steps are available can be controlled by Curry's search primitives [21]. Note that this example shows two useful applications of function patterns: the possibility to define abstractions for complex collections of patterns (via operation `evalTo`) and the ability to specify transformations at arbitrary positions inside an argument (via operation `replace`). The latter technique can be also exploited to formulate queries on expressions. For instance, the operation

```
varInExp (replace c p (Var v)) = v
```

non-deterministically returns a variable occurring in an expression. One can easily extract all variables occurring in an expression, by wrapping this operation with search primitives like `findall` [21]. □

Thus, function patterns are handy to provide executable high-level definitions of complex transformation tasks and queries on tree-like structures. Further examples of this kind (which we omit due to space limitations) are transformations and queries of XML terms.

## 5 Implementation

The transformational definition of the meaning of function patterns (Definition 3) does not lead to a constructive implementation since it might generate an infinite set of program rules. Any execution of a program, though, would make use of only a finite subset of these rules. In this section, we show a specialization of this transformation that, under suitable assumptions discussed later, enumerates the results of all the program rules that would be used in a specific execution of a program. These rules can be determined only at run time. This approach is easily integrated into existing implementations of functional logic languages, e.g., the Curry programming environment PAKCS [20]. We present some benchmarks of our implementation of this approach.

To integrate function patterns into existing implementations of functional logic languages, we eliminate the function patterns from the left-hand sides and move their functionality into the conditional part by means of a new *function pattern unification operator* "`=:<=`". The following transformation formalizes this process:

**Definition 4.** Let $P$ be a stratified functional logic program with function patterns. The *function pattern elimination* function $elim$ maps each rule into a rule without function patterns as follows:

$$elim(f\ t_1 \ldots t_n \mid c = r) =$$
$$\begin{cases} elim(f\ t_1 \ldots t_{i-1}\ x\ t_{i+1} \ldots t_n \mid t_i \texttt{ =:<= } x \texttt{ \& } c = r) & \text{if } t_1, \ldots, t_{i-1} \in \mathcal{T}(\mathcal{C}, \mathcal{X}), \\ & t_i \text{ contains functions}, \\ & x \text{ fresh variable}; \\ f\ t_1 \ldots t_n \mid c = r & \text{otherwise} \end{cases}$$ □

For instance, $elim$ maps rule (*last2*) to

```
last ys | xs++[x] =:<= ys = x  where xs,x free          (last3)
```

i.e., the pattern variables `xs` and `x` become logic variables in the transformed program. Their specific status will be used in the implementation of "`=:<=`".

It remains to implement the operator "`=:<=`". Its semantics is determined by Definition 3, i.e., the left argument must be evaluated to a constructor term that is finally matched against the right argument. This must be done with some care since the computation space of the left argument may be infinite. For instance, this situation occurs with the rule (*last3*). Consider again the computation of `last [failed,2]`. There are infinitely many evaluations of `xs++[x]` to a constructor term. However, among these evaluations every list with more or less than two elements cannot match `[failed,2]`.

To handle this situation, the evaluation of the function pattern by "`=:<=`" is demand-driven. This means that the function pattern is evaluated to a head normal form that is compared with the structure of the right argument and is further evaluated only if necessary. The details of function pattern unification follow.

To evaluate $e_1$ `=:<=` $e_2$:

1. Evaluate $e_1$ to a head normal form $h_1$
2. If $h_1$ is a variable: bind it to $e_2$
3. If $h_1 = c\, t_1 \ldots t_n$ (where $c$ is a constructor):
   (a) Evaluate $e_2$ to a head normal form $h_2$
   (b) If $h_2$ is a variable: instantiate $h_2$ to $c\, x_1 \ldots x_n$ ($x_1, \ldots, x_n$ are fresh variables) and evaluate $t_1$ `=:<=` $x_1$ `&` $\ldots$ `&` $t_n$ `=:<=` $x_n$
   (c) If $h_2 = c\, s_1 \ldots s_n$: evaluate $t_1$ `=:<=` $s_1$ `&` $\ldots$ `&` $t_n$ `=:<=` $s_n$
   (d) Otherwise: fail

Obviously, this implements the evaluation of the left argument of "`=:<=`" to a constructor term that is matched against the right argument. Since the evaluation of the left argument is interleaved with the matching, the search space of the evaluation of `xs++[x] =:<= [failed,2]` using the rule (*last3*) is finite (due to the failure in case 3d).

So far, we have only described the evaluation and binding of function patterns. However, the semantics of Definition 3 requires also the linearization of the evaluated function pattern combined with the addition of a strict equality constraint. One could consider integrating the linearization with the evaluation of "`=:<=`" in step 3: if the left argument is evaluated to the constructor-rooted term $c\, t_1 \ldots t_n$, we could replace multiple occurrences of a variable by new variables and generate strict equality constraints that are solved after the variables have been bound. Unfortunately, this method would be incorrect according to the semantics of Definition 3, since some repeated occurrences of a variable could be erased before the end of the evaluation. For instance, consider the following program:

```
k0 x = 0
pair x y = (x,y)
f (pair (k0 x) x) = 0
```

The meaning of `f` is equivalent to

```
f (0,x) = 0
```

by Definition 3. Consequently, `f (0,failed)` should evaluate to `0`. In the evaluation of `pair (k0 x) x =:<= (0,failed)`, the left argument is reduced to the constructor-rooted term `(k0 x,x)` where the variable `x` occurs twice. If we replace the first occurrence by `y` and generate the strict equality `y=:=x`, eventually we have to solve the

strict equality `y=:=failed` which causes the failure of the complete evaluation. Thus, the generation of strict equalities for the linearization of function patterns is a dynamic property. We have to keep track of the variables in function patterns that occur in the evaluated term. We can do this by marking the pattern variables that appear in the evaluated function pattern (i.e., in step 2). In this case the generated strict equality constraints are only executed when both involved variables have been marked during the evaluation of "`=:<=`". Thus, we obtain an incremental implementation of matching with function patterns conforming to the semantics specified by Definition 3.

Since the checking for multiple variable occurrences and the demand-driven generation of strict equality constraints for the involved variables might consume a considerable amount of time during function pattern unification, it is reasonable to optimize this part. Therefore, we have also implemented a second function pattern unification operator "`=:<<=`", which behaves like "`=:<=`" but does not check for multiple occurrences of variables in evaluated function patterns. It is safe to replace "`=:<=`" by the more efficient operation "`=:<<=`" if all the evaluations of the function pattern are linear. For instance, if a function pattern is linear and all the involved operations (i.e., also the ones that might be indirectly called) have rules with right-linear sides, then one can safely replace "`=:<=`" by "`=:<<=`". This is the case for our definition of `last` with rule (*last2*).

We formalize the correctness of our implementation as follows. For the sake of simplicity, we consider only unary functions, which typically are the only functions defined by a function pattern. *Let $R$ be a TRS, $m = f\,p \to r$ a rule defined by a function pattern and $m' = f\,x \mid p\texttt{=<:=}x \to r$ the transformed rule, where $x$ is a fresh variable. For all terms $t$ and $u$, $f\,t \to u$ in $R \cup \{m\}$ iff $f\,t \to u$ in $R \cup \{m'\}$.* The proof is simple except for the following claim: *In $R$, for all terms $p$ and $t$, $p\texttt{=<:=}t$ iff there exists a constructor term $l$ and a substitution $\sigma$ such that $p \overset{*}{\leadsto} l$ and $\sigma(l) = \sigma(t)$*. A rigorous proof of this result hinges on a formalization of our pattern unification algorithm that goes beyond the scope of this paper. In particular, the implementation requires a complete strategy to ensure that any constructor term $l$ such that $p \overset{*}{\leadsto} l$ is computed.

We have implemented function patterns in the Curry programming environment PAKCS [20]. This environment includes a compiler from Curry into Prolog [7], which we used for our benchmarks. The implementation is based on the ideas sketched above. Rules with function patterns are transformed into standard rules by putting the calls to "`=:<=`" in the condition part. Although function patterns aim at expressiveness rather than efficiency, we also show a few benchmarks where execution differences between function patterns and traditional strict equality are substantial. The benchmarks refer to the examples in this paper and are executed with strict equality ("`=:=`"), general function patterns ("`=:<=`"), and linear function patterns ("`=:<<=`"). The following table shows the execution results on a Pentium-M (1.6GHz) (all the times, in milliseconds, are the average of ten executions):[4]

[4] The operation `inc x n` increments `n` times `x` by 1. All the other operations are defined either in the standard prelude or in this paper. `simplify*` is the repeated application of the operation `simplify` to a large term with many opportunities for simplification, and `varsInExp` extracts all variables from a large term based on the operation `varInExp`.

| Expression: | "=:=" | "=:<=" | "=:<<=" |
|---|---|---|---|
| `last (take 10000 (repeat failed) ++ [1])` | no solution | 380 | 250 |
| `last (map (inc 0) [1..2000])` | 20900 | 90 | 60 |
| `lengthUpToRepeat ([1..50]++[1]++[51..])` | $\infty$ | 200 | 200 |
| `simplify*` | 1200 | 1080 | 690 |
| `varsInExp` | 2240 | 1040 | 100 |

As one can see, the specialization of matching linear function patterns with "`=:<<=`" can improve the efficiency considerably. Further improvements can be obtained by specializing the function patterns at compile time. For this purpose, we define an auxiliary operation

```
evalFP fp x e | fp =:<= x = e
```

Now, consider again the definition of `last`. Using `evalFP`, we can transform rule (*last3*) into (here, we omit the declaration of logic variables by where-clauses):

```
last zs = evalFP (xs++[x]) zs x                                    (last4)
```

According to Definition 3, the argument `(xs++[x])` must be evaluated by narrowing before it is matched against `zs`. Since there are two possible narrowing steps for `(xs++[x])`, we can replace the latter rule by:

```
last zs = last1 zs ? last2 zs
last1 zs = evalFP [x] zs x              ⟹  last1 [x] = x
last2 zs = evalFP (y:(ys++[x])) zs x
                ⟹  last2 (y:zs) = evalFP (ys++[x]) zs x
```

Since "`evalFP (ys++[x]) zs x`" is a variant of the right-hand side of rule (*last4*), with a folding step, we replace it by `last zs` and obtain the final definition

```
last zs = last1 zs ? last2 zs
last1 [x] = x
last2 (y:zs) = last zs
```

This is a functional logic program without function patterns. If we execute our previous benchmarks with this transformed program, we obtain the following results:

| Expression: | Transformed `last` |
|---|---|
| `last (take 10000 (repeat failed) ++ [1])` | 120 |
| `last (map (inc 0) [1..2000])` | 30 |

The speedup by a factor of 2 shows the advantages of this transformation. The specialization of a program rule with function patterns is similar to the partial evaluation of functional logic programs [2]. As such, it is difficult to predict whether a rule will yield a finite set of specialized rules and/or these rules will execute more efficiently then the transformation. The setting of function patterns differs from partial evaluation so that existing results and techniques cannot be directly applied. This demands for the development of new techniques — an interesting topic for future work.

## 6 Related Work

Although the idea to allow arbitrary user-defined functions in patterns is new in the context of functional logic languages, there exist many approaches to improve pattern

matching in declarative languages. Here we discuss the ones which have a closer relation to our work.

In functional logic languages, the considered kinds of patterns are usually constructor terms. Exceptions are languages like SLOG [12] or ALF [15], which allow functions in patterns that are useful to simplify terms before narrowing them. However, these languages are based on strict evaluation strategies and require the termination of the underlying rewrite system. Most other work in functional logic programming related to pattern matching considers only constructor patterns and concentrates mainly on sophisticated matching strategies in order to reduce the search space of narrowing computations (see [5, 16] for surveys).

Also in purely logic languages, patterns are constructor terms, and pattern matching is generalized to unification. An exception is constraint logic programming [23], where evaluable functions over constraint domains are allowed in patterns. However, they do not play any role in the pattern matching process since they are usually compiled into the right-hand side and passed to a separate constraint solver. Thus, most of the related work has been done for purely functional languages, as we will discuss next.

Context patterns [26], proposed for the functional language Haskell, are most closely related to our approach. The motivation for context patterns is somehow similar to the introduction of function patterns, since context patterns support the definition of functions based on the matching of subterms at an arbitrary depth. For instance, our `last` example can be defined with context patterns as

```
last (c [x]) = x
```

Here, `c` denotes a *context*, i.e., a term with a hole that is filled with the argument `[x]`. Similarly to function patterns, context patterns are useful to define queries and transformations over complex structures with a relatively small effort. However, due to the underlying functional base language, context patterns are more restricted. The holes in a context pattern are matched in a top-down left-to-right traversal against the actual argument and only the first match is taken. The author argues that this behavior, although incomplete, fits well into the framework of functional programming. Actually, he writes that a "non-deterministic approach would fit better in a integrated functional-logic language like Curry." It is interesting to note that the functional logic setting allows also to omit some other restrictions of context patterns, like the strict order of the traversal.

First-class patterns [28] are an approach to treat patterns as first-class objects (by considering patterns as functions of type "`a -> Maybe b`") in order to build abstractions for patterns and support user-defined strategies for pattern matching. This covers one aspect of function patterns in a purely functional setting, but the definition of operations with first-class patterns is rather clumsy even after introducing a specific language extension to support syntactic sugar for first-class patterns.

Transformational patterns [11] are another extension that supports the inclusion of user-defined functions in patterns. These functions are applied to the actual arguments before pattern matching in order to support a different view of the actual data (whose structure might be hidden in an abstract data type). This is orthogonal to our approach, in which functions are formal parameters that are evaluated and matched against the actual parameters.

Other related works include approaches to simplify writing code for term traversals. For instance, [24] shows a technique to support generic term traversals by defining small code pieces for each data type (comparable to the operation `replace` of Example 2) from which general functions to transform and query data structures can be derived. Although this technique leads to generic and efficient programs for manipulating trees, it does not have the generality of function patterns that can be also used to specify complex conditions on data structures (compare definition of `last` and Example 1).

## 7 Conclusions

We have proposed extending functional logic languages with function patterns. We have defined their semantics by transformation into traditional programs and shown that their implementation can be obtained by a specific unification procedure. Function patterns are advantageous because they evaluate conditions on actual arguments more lazily and thus avoid some known problems of strict equality. Moreover, they allow the high-level programming of queries and transformations of complex structures and support new abstractions of patterns.

This extension is specific to integrated functional logic languages since purely logic languages do not support evaluable functions and purely functional languages do not support nondeterminism and function inversion. The versatility and ease of implementation of function patterns show that an integrated functional logic language is an excellent environment for building high-level abstractions.

In future work, we plan to develop techniques to partially evaluate programs with function patterns at compile time to improve their efficiency. It might be interesting to develop specific calculi to support reasoning directly about programs with function pattens instead of using the transformational approach defined in this paper.

## References


1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation*, Vol. 40, No. 1, pp. 795–829, 2005.
2. M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, Vol. 20, No. 4, pp. 768–844, 1998.
3. S. Antoy. Optimal Non-Deterministic Functional Logic Computations. In *Proc. International Conference on Algebraic and Logic Programming (ALP'97)*, pp. 16–30. Springer LNCS 1298, 1997.
4. S. Antoy. Constructor-based Conditional Narrowing. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, pp. 199–206. ACM Press, 2001.
5. S. Antoy. Evaluation Strategies for Functional Logic Programming. *Journal of Symbolic Computation*, Vol. 40, No. 1, pp. 875–903, 2005.
6. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000.
7. S. Antoy and M. Hanus. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. International Workshop on Frontiers of Combining Systems (FroCoS'2000)*, pp. 171–185. Springer LNCS 1794, 2000.

8. S. Antoy and M. Hanus. Functional Logic Design Patterns. In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, pp. 67–87. Springer LNCS 2441, 2002.
9. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
10. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pp. 243–320. Elsevier, 1990.
11. M. Erwig and S. Peyton Jones. Pattern Guards and Transformational Patterns. *Electronic Notes in Theoretical Computer Science*, Vol. 41, No. 1, 2000.
12. L. Fribourg. SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 172–184, Boston, 1985.
13. E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel LEAF: A Logic plus Functional Language. *Journal of Computer and System Sciences*, Vol. 42, No. 2, pp. 139–185, 1991.
14. 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*, Vol. 40, pp. 47–87, 1999.
15. M. Hanus. Compiling Logic Programs with Equality. In *Proc. of the 2nd Int. Workshop on Programming Language Implementation and Logic Programming*, pp. 387–401. Springer LNCS 456, 1990.
16. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, Vol. 19&20, pp. 583–628, 1994.
17. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pp. 80–93, 1997.
18. M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pp. 47–62. Springer LNCS 1753, 2000.
19. M. Hanus. High-Level Server Side Web Scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pp. 76–92. Springer LNCS 1990, 2001.
20. M. Hanus, S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2004.
21. M. Hanus and F. Steiner. Controlling Search in Declarative Programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, pp. 374–390. Springer LNCS 1490, 1998.
22. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8). Available at `http://www.informatik.uni-kiel.de/~curry`, 2003.
23. J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. of the 14th ACM Symposium on Principles of Programming Languages*, pp. 111–119, Munich, 1987.
24. R. Lämmel and S.L. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI'03)*, pp. 26–37. ACM Press, 2003.
25. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pp. 244–247. Springer LNCS 1631, 1999.
26. M. Mohnen. Context Patterns in Haskell. In *Implementation of Functional Languages*, pp. 41–57. Springer LNCS 1268, 1997.
27. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
28. M. Tullsen. First class patterns. In *2nd International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pp. 1–15. Springer LNCS 1753, 2000.