Application patterns

Nikolaas N. Oosterhof, Philip K. F. Hölzenspies, and Jan Kuper
{n.n.oosterhof, p.k.f.holzenspies}@student.utwente.nl
jankuper@cs.utwente.nl

University of Twente
Department of Computer Science
P.O.Box 217
7500 AE Enschede
The Netherlands

Abstract

We present a generalization of patterns as used in definitions in functional languages, called application patterns. They consist of a function applied to arguments. While matching such a pattern against an actual argument, inverse functions are used to find the binding of variables to values. Application patterns are universal in the sense that they include list, tuple, algebraic and `n+k` patterns.

1 INTRODUCTION

Pattern matching plays an important role in functional programming. By using patterns the readability of a definition as well as the structure of a program may be improved. Standard patterns include variables, constants, the wildcard pattern, patterns for tuples, lists, algebraic constructors, etc. Such patterns match against the structure of an argument, in such a way that lazy evaluation remains possible.

In addition, there are `n+k`-patterns. The nature of these is somewhat different from the ones above, they do not match against the syntactical structure of an actual argument, but against its semantics (assuming the standard implementation of natural numbers). A pattern `n+k`, where `k` is a constant, matches against an actual argument *a*, if *a* can be considered as the result of applying the *function* $\lambda x.x+\mathtt{k}$ to some value *b*. If so, then `n` is bound to *b*. Clearly, *b* can be calculated by applying the *inverse function* $\lambda x.x-\mathtt{k}$ to *a*.

We call a pattern such as `n+k` an *application pattern* since it consists of a function (+) applied to some arguments (`n` and `k`). In this paper we describe application patterns in general, and we show that known (compound) patterns can be expressed as application patterns. Furthermore, we present an algorithm which matches an application pattern against an actual argument, and which binds variables to values.

We stipulate that matching an application pattern like `f x y` against an actual argument is based on the *semantics* of a function and an argument, i.e. the values for `x` and `y` are found by applying *inverse functions* of `f` to the actual argument (assuming that these inverses exist).

Patterns are subject of several discussions (see e.g. the Haskell mailing list), but as far as the authors of the present papers are aware, no proposals have been made in the direction of application patterns. One paper that we know of is by Tullsen (2000), which in a restricted sense uses inverse functions. In that paper inverses of algebraic constructors are used, but not for functions in general. Besides, the content of Tullsen's paper is aimed at developing "a simplification of patterns that make them first class language constructs".

In Gofer some work is done in matching patterns of generalizations of the form `c*n+k` (Tony Davy, n.d., as cited in Jones, 1991). That is a specific case of what we discuss in general.

Quite some work has been done in the field of automatic computation of a function's inverse (esp. Abramov and Glück (2002) and Plasmeijer et al. (2005)), but that is not a topic of the present paper. In this paper we assume that it is the responsibility of the programmer to make sure that the necessary inverses do exist, either in the standard library or in the program being compiled.

2 GENERAL FORM OF APPLICATION PATTERNS

Suppose

```
double_a :: [char] -> [char]
```

is a function which should double a string when all characters in that string are equal to the character `'a'`, return the string itself when all characters are equal to any other character, and yield the empty string otherwise. Thus,

```
double_a "aaa"  ⟹ "aaaaaa"
double_a "bbb"  ⟹ "bbb"
double_a "abc"  ⟹ ""
```

One possible way to define the function `double_a` is as follows:

```
double_a []     = []
double_a (x:xs) = x:xs ++ x:xs, if and (map (='a') (x:xs))
                = x:xs        , if and (map (=x) xs)
                = ""          , otherwise
```

However, observe that the argument is tested for consisting of equal characters. Thus, the question is whether the argument can be considered as a result of a *repeat* function. Then, using *application patterns*, `double_a` could also be defined in a shorter and better readable way:

```
double_a (rep 'a' n) = rep 'a' (2*n)
double_a (rep  x  n) = rep  x  n
double_a _           = ""
```

In this definition `rep` is a standard function which returns a list of copies of its first argument, where the length of this list is indicated by the second argument of `rep`.

Calling the function `double_a` with actual argument `"aaa"`, the first pattern, `rep 'a' n`, matches `"aaa"`, and `n` is bound to `3`. The string `"bbb"` does not match the pattern in the first clause and thus the second clause is tried, leading to the bindings of x to 'b' and n to 3. Note that the case of the empty string is automatically covered by the application pattern.

Application patterns also allow for nested patterns, as in

```
double_a (rep 'a' (n+1)) = ... n ...
```

The general form of an application pattern is

$$\mathit{fun}\ \mathit{pat}_1\ \mathit{pat}_2\ \cdots\ \mathit{pat}_n$$

where *fun* is an existing function, and each pat_i can be any pattern, including an application pattern. Note that algebraic patterns, including lists and tuples, are just a specific case of application patterns.

In the general form of an application pattern, the function *fun* may be a predefined function, or defined by the programmer. However, in order for the pattern to match, the function *fun* should have appropriate inverses in the sense described below. Then the values to which the variables in the pattern will be bound are found by choosing the adequate inverse function and applying that to the actual argument. In the example above, *two* inverses of the function rep are needed:

- one should find n, *given* that the character of which the actual argument is of a specific form (in this case it should be an 'a'),
- the other should reconstruct *both* x and n from the actual argument (if possible).

Clearly, there is a third variant of an inverse function for rep, not needed in the definition of double_a:

- find the character x, given that the length of the actual argument is given.

3 INVERSE FUNCTIONS

Let f be a function of type $A \rightarrow B$. It is well known that if f is injective, then the inverse f^{-1} exists and is of type $B \rightarrow A$. However, it is not so clear how to deal with possible inverse function(s) of f in case f is a function of several arguments as in the case of rep above. Of course, rep is of type A -> (Nat->[A]), and thus the standard inverse would be of type (Nat->[A]) -> A. But in the example given above two *different* inverses of rep are needed, the first yielding a value of type Nat, the second of type (A,Nat).

Thus, we need a *generalized* form of inverse functions. Consider a function

$$f :: A_1 \rightarrow A_2 \rightarrow \cdots \rightarrow A_n \rightarrow B$$

with

$$f\ x_1\ x_2\ \cdots\ x_n = \mathit{expr}$$

Let $i_1,\ldots,i_k$ be a sublist of the indices $1,\ldots,n$, and let $j_1,\ldots,j_m$ be the remaining indices, where both sublists of indices are in increasing order. We call a function $f^c_{i_1,\ldots,i_k}$ of type

$$f^c_{i_1,\ldots,i_k} :: A_{j_1} \to \cdots \to A_{j_m} \to (A_{i_1},\ldots,A_{i_k}) \to B$$

such that

$$f^c_{i_1,\ldots,i_k}\, x_{j_1}\, \cdots\, x_{j_m}\, (x_{i_1},\ldots,x_{i_k}) = f\, x_1\, x_2\, \cdots\, x_n$$

a *cousin* with respect to $i_1,\ldots,i_k$ of the function f. Thus the only difference between a function f and a cousin $f^c_{i_1,\ldots,i_k}$ is that the latter groups the arguments $x_{i_1},\ldots,x_{i_k}$ into a tuple and takes that tuple as its last argument.

A *generalized inverse* of f with respect to $i_1,\ldots,i_k$, denoted as $f^{-1}_{i_1,\ldots,i_k}$, is a function with type

$$A_{j_1} \to \cdots \to A_{j_m} \to B \;\to\; (A_{i_1},\ldots,A_{i_k})$$

such that

$$f^{-1}_{i_1,\ldots,i_k}\, x_{j_1}\, \cdots\, x_{j_m}\, y \;=\; (x_{i_1},\ldots,x_{i_k})$$

if and only if

$$f^c_{i_1,\ldots,i_k}\, x_{j_1}\, \cdots\, x_{j_m}\, (x_{i_1},\ldots,x_{i_k}) = y$$

i.e. if and only if

$$f\, x_1\, x_2\, \cdots\, x_n = y$$

For a generalized inverse $f^{-1}_{i_1,\ldots,i_k}$ of f it holds that

$$f^{-1}_{i_1,\ldots,i_k}\, x_{j_1}\, \cdots\, x_{j_m} \;=\; (f^c_{i_1,\ldots,i_k}\, x_{j_1}\, \cdots\, x_{j_m})^{-1}$$

Thus, the generalized inverse function $f^{-1}_{i_1,\ldots,i_k}$ yields values for the variables on the positions $i_1,\ldots,i_k$, given that the parameters on the other positions are known. Clearly, since functions need not be injective, not all generalized inverses will exist for all sequences of indices.

As described in the example of `double_a` in section 2, generalized inverse functions are used to resolve application patterns that are used in a program. It is the responsibility of the programmer to make sure that the relevant inverse functions do exist. If some inverse function is not present in the prelude of the system, or if it can not be generated automatically, the programmer has to write it himself. For that we introduce the *backtic notation*: the notation $f^{-1}_{i_1,\ldots,i_k}$ for a generalized inverse function can be written as

```
f`[ i₁−1,⋯,iₖ−1 ]
```

where each index is decremented by one, since in computer science it is custom to start counting at zero instead of one.

This notation is meant for the programmer to write his own definitions of generalized inverse functions. Hence `f`[` $i_1-1$`,⋯,`$i_k-1$`]` should be considered as a (systematic) *name* in itself, i.e. the backtick is *not* an operator which yields inverse functions.

In the case of the pattern `rep x n`, as in (see section 2)

```
double_a (rep x n) = ...
```

the different variants of these inverses may be defined as follows (for an alternative definition, see section 4):

```
rep`[0,1] :: [A] -> (A, Nat)
rep`[0,1]  []    = error "type of list elements ambiguous"
rep`[0,1] (x:xs) = (x, #(x:xs))       , if and (map (=x) xs)

rep`[0] :: Nat -> [A] -> A
rep`[0] n  []    = error "type of list elements ambiguous"
rep`[0] n (x:xs) = x                  , if and (map (=x) xs)
                                        /\ #(x:xs) = n

rep`[1] :: A -> [A] -> Nat
rep`[1] a xs = #xs                    , if and (map (=a) xs)
```

Note that definitions for both `` rep`[0] `` and `` rep`[1] `` can be derived automatically when `` rep`[0,1] `` is given, e.g.

```
rep`[0] n (rep m x) = x   , if m=n
```

In this definition, bindings for `m` and `x` are yielded by `` rep`[0,1] ``.

Note that these definitions do not contain an `otherwise` guard. The role of `otherwise` would be to express that the inverse is *not defined*, and we choose to leave it out. By this construction an application pattern can be *refutable*. In section 6 these functions will be rewritten so that failure to meet with any of the given guards will result in a pattern miss.

With the above definitions of the various inverses of `rep`, the following uses lead to the intended answers:

```
rep`[0,1] "aaa"     ⟹ ('a',3)
rep`[1]   'a' "aaa" ⟹ 3
rep`[0]   3 "aaa"   ⟹ a
```

As an example of a function for which not every generalized inverse exists, consider the function

```
minus x y = x - y
```

Clearly, the inverse function `` minus`[0,1] `` does not exist, since any given number `n` may be decomposed in an infinite number of ways into two numbers whose difference is `n`. However, the inverses for both individual positions `` minus`[0] `` and `` minus`[1] `` do exist.

For many other functions (partial) inverses can be defined, e.g. for `+`, `-`, `*`, `/`, `^`, `e^`, `ln`, `sin`, `cos`, `tan`, `arcsin`, `arccos`, `arctan`, `prime`, `fac`, `++`, `index`, `rep`, `itoa`, `code`, `decode`, `lines` and `unlines`. Moreover, for constructors of algebraic types the inverses follow trivially from the type definition in which they are introduced.

4 WHERE CLAUSES

In the above, patterns were used solely on parameter positions in function definitions. Here we discuss how to use them in `let` and `where` clauses. Consider an alternative definition for `rep`[0,1]`

```
rep`[0,1] [x]        = (x,1)
rep`[0,1] (x:y:tail) = (x,n)    , if x=y /\ y=z
                     where
                       rep z (n-1) = y:tail
```

In the `where` clause the application pattern `rep z (n-1)` will bind `z` and `n` to their values such that they can be used in the defining expression. This pattern needs the generalised inverse function `rep`[0,1]` in order to be resolved. That is precisely the function which is defined here. Thus, this definition of `rep`[0,1]` is recursive.

However, the use of application patterns within `where` clauses is ambiguous. Instead of reading `rep z (n-1)` as a pattern, it might as well be read as a local definition of `rep`.

Therefore we introduce a syntactical means to indicate which parameters get their value from the context. In the case of the pattern `rep z (n-1)` the only ambiguity arises for the role of `rep`. We write

```
... where
      ^rep z (n-1) = ...
```

to indicate that `rep` is an existing function. The expression thus has to be read as a pattern by which the variables `z` and `n` have to be bound. The same ambiguity is present in `n+k` patterns.

The `^` notation allows for the possibility to bind other parameters in a pattern by a definition in the context, as in

```
... where
      z             = ...
      ^rep ^z (n-1) = ...
```

Here, the `z` in the pattern gets its value from the pattern in the global function definition, such that only the variable `n` has to be bound by matching the pattern to the right hand side. Note that in nested patterns, such as `n-1` in the example above, no ambiguity can arise.

With respect to operators the ^ prefix notation would lead to undesirable constructions as in `x^:xs` and `n^+5`. Therefore we propose that infix definitions of these operators require special syntactic sugar, so that in their ordinary use (binding variables) the ^ character can be omitted. For example, in Haskell this ambiguity for operator symbols is solved by putting brackets around the expression.

5 SOME EXAMPLES

Using the application pattern, function definitions become more readible if some trivial operation must be performed on an argument first, as in

```
f₁ (sin alpha) = ... alpha ...
f₂ (2*n)       = ... n ...
f₃ (itoa s)    = ... s ...
f₄ (ln x)      = ... x ...
```

For example, if f_1 is applied to an actual argument `a`, the function `sin‘[0]`—that can be pre-defined as the arcsin function—is applied to `a` and the result bound to `alpha`. Note that f_1 is only defined for values between `-1` and `1`, inclusive. Likewise, the functions f_2, f_3 and f_4 use, when applied to an actual argument, the inverse functions `*‘[1]`, `itoa‘[0]` and `ln‘[0]` in order to bind `n`, `s` and `x`, respectively. These inverse function definitions are trivial, by the use of division, `atoi` and exp functions.

As another example, consider the function `upperLeft`, which finds the upper left corner of the rectangle determined by a list of points in the two dimensional plane.

```
upperLeft :: [(Float, Float)] -> (Float, Float)
upperLeft (zip (xs,ys)) = (min xs, max ys)
```

In this example the pattern `zip (xs,ys)` splits the actual argument (a list of (x,y)-coordinates) into a list of x-coordinates `xs`, and a list of y-coordinates `ys`.

Yet another application is simple string parsing. Let `join2` and `join3` be functions that join two and three lists, respectively.

```
join2 x y   = x ++ y
join3 x y z = x ++ y ++ z
```

The generalized inverses for `join2` can be defined as

```
join2‘[0] y s
  = reverse q, if y = reverse p
  where
    (p, q) = split (#y) (reverse s)

join2‘[1] x s
  = q, if p = x
  where
    (p, q) = split (#x) s
```

Thus, for example,

```
join2‘[0] "de" "abcde"  ⟹ "abc"
join2‘[1] "abc" "abcde" ⟹ "de"
```

Moreover, for `join3` generalized inverses `join3‘[0,2]` and `join3‘[1]` can be defined as well, but we will omit them here for brevity.

Note that `join2‘[0]` might also have been defined using the application pattern, so that the symmetry between the definitions of `join‘[0]` and `join‘[1]` becomes more apparant.

```
reverse‘[0] z = reverse z

join2‘[0] x s
  = q, if p = x
  where
    (reverse p, reverse q) = split (#x) (reverse s)
```

One nice application of `join2` and `join3` would be an extended use of list patterns. For example, a function that would ‘parse’ a pair of numbers can now be specified as

```
parsePair :: [Char] -> (Nat, Nat)
parsePair ("(" ++ itoa x ++ "," ++ itoa y ++ ")") = (x, y)
```

Clearly, the use of `++` does not work directly, but using `join2` and `join3` this pattern can automatically be rewritten as

```
parsePair (join2 "("
                 (join3 (itoa x)
                        ","
                        (join2 (itoa y)
                               ")"       )))
  = (x, y)
```

so that `x` and `y` are bound using application patterns.

Note, however, that in general this rewriting causes that the “first” match is chosen for string arguments, i.e. the first occurrence of a separator symbol such as a comma is chosen to match the pattern.

6 TRANSLATION ALGORITHM

In this section we show how to translate application patterns to traditional patterns, in combination with guards and where clauses.

First we remark that, since application patterns must be refutable, the translation algorithm first automatically rewrites an inverse function definition so that its result is of the `maybe` type:

```
data Maybe a = Just a | Nothing
```

This rewriting $*$ is informally specified by

$$\begin{array}{lcl} \texttt{f`[a}_1\texttt{,}\cdots\texttt{,a}_k\texttt{] x}_1 \cdots \texttt{x}_m & & \texttt{f`[a}_1\texttt{,}\cdots\texttt{,a}_k\texttt{] x}_1 \cdots \texttt{x}_m \\ \quad \texttt{= v}_1\texttt{, if g}_1 & & \quad \texttt{= Just v}_1\texttt{, if g}_1 \\ \quad\quad \vdots \quad\quad \vdots & \overset{*}{\Longrightarrow} & \quad\quad\quad \vdots \quad\quad \vdots \\ \quad \texttt{= v}_n\texttt{, if g}_n & & \quad \texttt{= Just v}_n\texttt{, if g}_n \\ & & \quad \texttt{= Nothing, otherwise} \end{array}$$

where the final `otherwise` clause on the right-hand side is omitted if the last guard $\texttt{g}_n$ on the left-hand side is `otherwise`.

For example, in the case of the definitions of `join2‘[0]` and `join2‘[1]` we get (note that here the `otherwise` clause is filled in):

```
join2‘[0] y s
  = Just (reverse q), if y = reverse p
  = Nothing, otherwise
  where
    (p, q) = split (#y) (reverse s)

join2‘[1] x s
  = Just q, if p = x
  = Nothing, otherwise
  where
    (p, q) = split (#x) s
```

We only describe the results of the algorithm on an example, we don't give the translation in detail. It will be described in more detail by Oosterhof (2005, Upcoming).

Suppose a function `f` is defined as follows (`x`, `y`, `v` and `w` are variables; c is a constant expression; e_0, e_1 and e_2 are expressions containing the indicated variables):

```
f (g x (h y c)) = e0(v,w)
                  where
                    ^k v w = e1(x,y)
f (u:us)          = e2(u,us)
```

The first clause contains three (nested) application patterns, using the functions `g`, `h` and `k`. Thus, these functions plus their appropriate inverses should exist.

First we concentrate on the first clause of `f`, the second clause is postponed until later. The translation starts with generating variables (say `p0` and `p1`) to represent the application patterns. The values to which the variables in the original pattern have to be bound, are then produced by applying the appropriate inverses of the functions g and h in the pattern to these newly introduced variables. This yields the *matching* variables m0 and m1. This is done in a where clause. That gives:

```
f p0 = e0(v,w)                , if m0 ~= Nothing
                                 ∧ m1 ~= Nothing
     where
       m0           = g`[0,1] p0
       Just (x,p1) = m0

       m1     = h`[0] c p1
       Just y = m1

       m2           = k`[0,1] e1(x,y)
       Just (v,w) = m2
```

Note that m0 is of the Maybe type, so first it should be clear that it is not Nothing, before its values can be extracted. Then its second value, p1, stands for the inner application pattern h y *c*, so the procedure is repeated for an inverse of h.

Note that the matching variable m2 may also be Nothing. We might add a guard m2~=Nothing, but that would mean that the decision whether an argument matches a pattern would depend on an undefined result in a where clause. We feel that this is the wrong choice. Hence, the last two lines in the definition above might be replaced by one line

```
Just (v,w) = k`[0,1] e1(x,y)
```

If the right hand side leads to an undefined result, this gives a run time error.

At this point we remark that instead of choosing h`[0] we also might have chosen h`[0,1]. In that case *two* values y and z would have been yielded in m1, and an extra guard would have been necessary to check whether z=c (compare the automatic generation of rep`[0] in section 3). This depends on the inverse functions that are available in the program, the essential point being that at least values for the formal parameters (in this case, only y) have to be yielded. Our algorithm chooses one of those variants of the inverse functions which at least yields values for the required variables in the pattern.

The second clause in the definition of f contains the standard pattern u:us. For consistency reasons, we translate such patterns to corresponding application patterns. In this case the obvious translation is

```
cons u us
```

and the relevant inverse of cons is defined as

```
cons`[0,1] xs = (hd xs, tl xs), if xs ~= []
```

Thus, the translation of the second clause of f becomes:

```
  f p = e2(u,us)              , if m ~= Nothing
      where
        m           = cons`[0,1] p
        Just (u,us) = m
```

However, since the translation of the first clause of `f` resulted in a clause with a variable pattern, the second clause became unreachable. Thus, all clauses in the definition of `f` have to be combined into one clause in the translated definition. Besides, a final `otherwise` clause that throws a runtime exception is added to indicate missing cases.

The translation of the total definition of `f` now becomes:

```
f p = e0(v,w)                 , if m0 ~= Nothing
                                 ∧ m1 ~= Nothing
    = e2(u,us)                , if m3 ~= Nothing
    = error "missing case" , otherwise
    where
      m0          = g`[0,1] p
      Just (x,p') = m0

      m1     = h`[0] c p'
      Just y = m1

      Just (v,w) = k`[0,1] e1(x,y)

      m3          = cons`[0,1] p
      Just (u,us) = m3
```

Clearly, while joining different clauses in the definition of `f` into one clause, one has to be careful to avoid name clashes. However, that is an administrative job, and not the most interesting part of the algorithm. Therefore we skip it here.

One topic that deserves special attention is to what extent the use of application patterns preserves lazyness. Since application patterns are based on semantic values and not on syntactic structure, it might be so that an argument has to be evaluated fully in order to decide whether it matches a given pattern or not. In the translation algoritm we presented here, we used the `Maybe` type to translate definitions using application patterns to definitons using inverse functions in a `where` clause. The outcome of this was compared to the value `Nothing`. Since a value of the form `Just` ... need not be evaluated fully in order to decide that it is not equal to `Nothing`, it is our impression that application patterns can be evaluated lazily. However, this is a point of future research.

7 FURTHER GENERALIZATION

A further generalization of application patterns may use them to extract *any* parameter from the actual argument, i.e. to yield bindings of parameters to values that are not necessarily extracted by using inverse functions. Then the programmer has the ultimate freedom to write his own patterns and choose those parts of the actual argument that he needs. For example, suppose a programmer wants to write a function `rep1` that takes a list as argument, and returns a list consisting of repetitions of the first element. The length of the returned list should be the same as the length of the argument.

$$\texttt{rep1 "abc"} \Longrightarrow \texttt{"aaa"}$$

Using application patterns, the function `rep1` may be defined as follows:

```
rep1 (foo x n) = rep x n
```

where *foo* is any name that suits the programmer, and which indicates that only the first element (bound to `x`) and the length (bound to `n`) of the actual argument are needed. The programmer then has the obligation to write the function `foo‘[0,1]`, called an *extraction function* since it need not be an inverse function:

```
foo‘[0,1] xs = (hd xs, #xs), if xs ~= []
```

Note that *foo* itself need not be an existing function.

Besides, a programmer may decide it is safe to assume that the actual argument is a proper result of an existing function, say `rep`, and only wants to bind the variable. Then the corresponding extraction functions of `rep` can shortly be defined as:

```
rep‘[0,1] (r:rs) = (r, 1+#rs)

rep‘[0] _ (r:_) = r

rep‘[1] _ rs = #rs
```

For semantic reasons, however, care has to be taken with giving names to functions in such cases.

One application is the extraction function

```
check‘[1] checkIt v = v, if checkIt v
```

that allows for guards in function arguments. For example, the power function for integers can be specified by

```
power b 0             = 1
power b (check (<0) x) = 1 / (power b (-x))
power b (x+1)         = b * power b x
```

where its definion’s three clauses show examples of an ordinary constant pattern, an application pattern with the extraction function `check‘[1]` and an application pattern that uses the inverse addition function `+‘[0]`, respectively.

8 FUTURE RESEARCH

The use of patterns described above suggests a style of "programming by equations", i.e. definitions take the form of general equations where solving such equations results in binding variables to values. One additional aspect of such equations is that the same variable may occur more than once in a pattern, and solving them may require a step by step approach. For example, applying a function like

```
f (x-a) ((a+y)*a) (a+x) (x+3) = x + y + 2 * a
```

to four actual arguments leads to the bindings of `a`, `x` and `y`. Here, first `x` has to be solved through the fourth argument. Then `a` can be solved through the first argument by using the value of `x`. Next, `y` can be solved through the second argument by using the value of `a`. Finally it must be checked whether the third argument equals `a+x`.

We have built a runtime pattern matcher that can already do this; it seems an interesting question whether this can be extended to compile time pattern matching. This would provide a general solution for such equation style programming. An inherent drawback of this approach, however, is that the guarantee of lazyness of in-order matching of the arguments is lost and becomes the sole responsibility of the programmer. Ways around this drawback are subject of further research also.

ACKNOWLEDGEMENTS

We thank Arjan Boeijink, Maarten Fokkinga and Joeri van Ruth for their valuable remarks.

REFERENCES

Abramov, S. A., & Glück, R. (2002). The universal resolving algorithm and its correctness: Inverse computation in a functional language. *Science of Computer Programming*, *43*(2–3), 193–229.

Jones, M. P. (1991, November). Gofer 2.21 release notes. Retrieved on July 7, 2005, from `http://www-i2.informatik.rwth-aachen.de/Teaching/Praktikum/SWPSS96/Gofer/rel221.dvi`.

Oosterhof, N. N. (2005). *Application patterns in functional languages*. Unpublished master's thesis, University of Twente, Enschede, The Netherlands. (Upcoming)

Plasmeijer, M., et al. (2005, July). (Personal communication)

Tullsen, M. (2000, January). First class patterns. In E. Pontelli & V. S. Costa (Eds.), *Practical aspects of declarative languages, second international workshop, padl 2000* (Vol. 1753, p. 1-15). Springer-Verlag.