Context Patterns, Part II

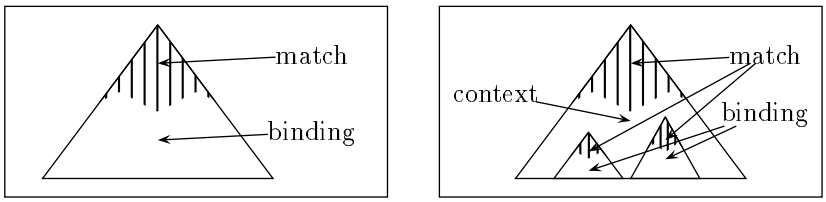
Markus Mohnen

Lehrstuhl für Informatik II, RWTH Aachen, Germany
mohnen@informatik.rwth-aachen.de

Abstract. Functional languages allow the definition of functions by pattern matching, which performs an analysis of the *structure of values*. However, the structures which can be described by such patterns are restricted to a fixed portion from the the root of the value.
Context patterns are a new non–local form of patterns, which allow the matching of subterms without fixed distance from the root of the value.
Typical applications of context patterns are functions which *search* a data structure for patterns and *transform* it by replacing the pattern.
In this paper we introduce a new construct called *extended context*, which allows the definition of transformational functions without superfluous repetition of the recursive search.

1 Introduction

This paper is the successor to "Context Patterns in Haskell" [Moh97]. That work described a new non–local kind of pattern which allows matching of subterms without fixed distance from the root of the value. The underlying observation is that standard patterns allow only the matching of a fixed region near the root of the structure. Consequently, the resulting bindings are substructures adjacent to the region (see Figure 1(a)). It is neither possible to specify patterns at a non–fixed distance (possibly far) from the root, nor to bind the context of such a pattern to a variable (see Figure 1(b)).



(a) Standard Pattern (b) Context Pattern

Fig. 1. Pattern Matching

Context patterns are a flexible and elegant extension of traditional patterns. Typical examples of functions using the increased expressive power are functions which search and transform data structures.

Transformations often replace all occurrences of a pattern without introducing new occurrences. Such function can traverse a data structure only once. However, the use of context patterns in combination with recursion imposed an additional repetition of the recursive search for those parts before the last match.

In this paper, we extend context patterns to overcome this deficiency: We introduce a new construct called *extended context*, which gives access to the unvisited part of a data structure. Recursive calls of a transformation can be placed to visit these parts only without revisiting the parts before the last match.

Furthermore, we give a formal definition of the static semantics of context patterns.

This paper is organised as follows. In Section 2 we introduce context patterns by example program and discuss the necessary modification to the approach in [Moh97]. Section 3 defines syntax and static semantics of context patterns. The implementation is described by a translation of context patterns to standard Haskell which is presented in Section 4. Some of the previous work in pattern matching and topics related to our approach is reviewed in Section 5. Section 6 concludes.

2 Context Patterns by Examples

To demonstrate the basic idea, we start with a toy example. Consider a function `initlast :: [a] -> ([a],a)` which splits a list into its initial part and its last element. Informally, we can describe an implementation as a single match:

> Take everything up to but not including a one–element list as initial part and the element of the list as last element.

However, we cannot express this match by using the standard patterns. Instead, the recursive search must be programmed explicitly:

```
initlast :: [a] -> ([a],a)
initlast []     = error "Empty list"
initlast [x]    = ([],x)
initlast (x:xs) = let (ys,l) = initlast xs in (x:ys,l)
```

Our extension consists of a single additional pattern called *context pattern*, with the syntax:

$$cpat \quad \rightarrow \quad var\ pat_1\ \ldots\ pat_k$$

A context pattern matches a value v if there exists a function f and values $v_1, \ldots, v_k$ such that pat_i matches v_i and $f\ v_1\ \ldots\ v_k$ is equal to v. Furthermore, this function f is a representation of a *constructor context* [Bar85], i.e. a constructor term with "holes". The representation consists of modelling the "hole" by the function arguments:

$$f = \lambda h_1 \ldots \lambda h_k . C[h_1, \ldots, h_k]$$

where C is a constructor context with k "holes", which imitates the shape of the value v. If the pattern matches the value, the function f is bound to the variable var.

In our example, we can reformulate `initlast` using context patterns in the following way:

```
initlast :: [a] -> ([a],a)
initlast []      = error "Empty list"
initlast (c [x]) = ((c []), x)
```

Applying `initlast` to a list $[a_1, \ldots, a_{n-1}, a_n]$ gives us the following bindings:

$$[\texttt{x}/a_n, \texttt{c}/\texttt{\textbackslash l->(}a_1\texttt{:}\ldots\texttt{(}a_{n-1}\texttt{:l)}\ldots\texttt{)}]$$

Hence, evaluation of the application `(c [])` on the right hand side. yields the initial part $[a_1, \ldots, a_{n-1}]$.

Transformations typically replace *all* occurrences of a pattern. We can easily do this by combining context patterns and (tail–)recursion. For instance, replacing all occurrences of the character `a` in a string can simply be done by the following function

```
rplc_a :: Char -> String -> String
rplc_a x (c 'a') = rplc_a x (c x)
rplc_a x s = s
```

The first rule searches for the first occurrence of the character `'a'` in the string. By the application `(c x)` a new string is created where this occurrence is replaced by the value of `x`. The recursive call of `rplc_a` then replaces all other occurrences until the context patterns fails and the result is returned in the second rule.

Although this technique is applicable in general, it imposes an additional overhead for the case that no new occurrences of the pattern are created for the argument of the recursive call: The portion of the string before the first occurrence of `a` is searched again by all recursive calls to `rplc_a`, although this is not necessary. For this example, this unnecessary searching results in quadratic complexity of `rplc_a`, although only linear complexity is needed.

To overcome this deficiency, we extend the approach. We assume that we have a purely transformational function, where the result has the same type as the input. To avoid the searching of the already searched part, we simply have to ensure that the recursive call is only applied to the unvisited subvalues of same type in the context. Since the context function does not allow access to these parts, we introduce another variant of the context function: Assume that we have a context pattern $c\ pat_1\ \ldots\ pat_k$. In addition to the context c with type $t_1 \to \cdots \to t_k \to t$ an *extended context* `c_e` of type $(t \to t) \to t_1 \to \cdots \to t_k \to t$ is available on the right hand side. The difference is that while c leaves all unvisited subvalues of type t in the context unchanged, `c_e` applies its functional argument to these subvalues: We have $c = (c\texttt{\_e}\ id)$. By using this extension, we can reformulate `rplc_a` in the following way:

```
rplc_a :: Char -> String -> String
rplc_a x (c 'a') = c_e (rplc_a x) x
rplc_a x s = s
```

Given the string `"snafu"`, we obtain the following bindings for the context and the extended context:

$$\texttt{c} = \lambda x.\texttt{'s':'n':}x\texttt{++"fu"}$$
$$\texttt{c\_e} = \lambda f.\lambda x.\texttt{'s':'n':}x\texttt{++}(f\ \texttt{"fu"})$$

Hence, the recursive call applies `rplc_a` to the string `"fu"` only.

In addition to the basic syntax of context patterns there a three extras features which are discussed in more detail in [Moh97]:

- *Context wildcards* (`_`) can be used to match contexts which are not used on the right hand side. For instance, to obtain the last element of a list, we can define:

```
last :: [a] -> a
last []      = error "Empty list"
last (_ [x]) = x
```

– *Guards* introduce additional (non-free) conditions during the recursive search. An example which uses this feature is an implementation of `takeWhile`

```
takeWhile :: (a->Bool) -> [a] -> [a]
takeWhile p (c (x:_) if (not (p x))) = c []
takeWhile p l = l
```

The first rule matches a context where the head of the remaining list is the first element which does not satisfy `p`. By replacing the remaining list by the empty list in the context, we obtain the longest prefix of elements satisfying `p`. The second rule is to handle those lists where all elements belong to this prefix.

– *Explicit types* at the patterns can be used to avoid underspecified context patterns.

3 Syntax and Static Semantics

The syntax of Haskell's patterns is shown in Figure 2(a) (taken from [HPW92, pp. 17–18]). For simplicity, we omit as–patterns, irrefutable patterns, infix patterns, tuple patterns, unit patterns, and $n + k$ patterns. Our extension is in Figure 2(b).

There is one small conflict which arises with this extension: The definition

$$\texttt{let x (y:ys) =}\ e_1\ \texttt{in}\ e_2$$

can be either a function definition for `x` using the pattern `y:ys`, or a context pattern. In these cases, function definitions are preferred.

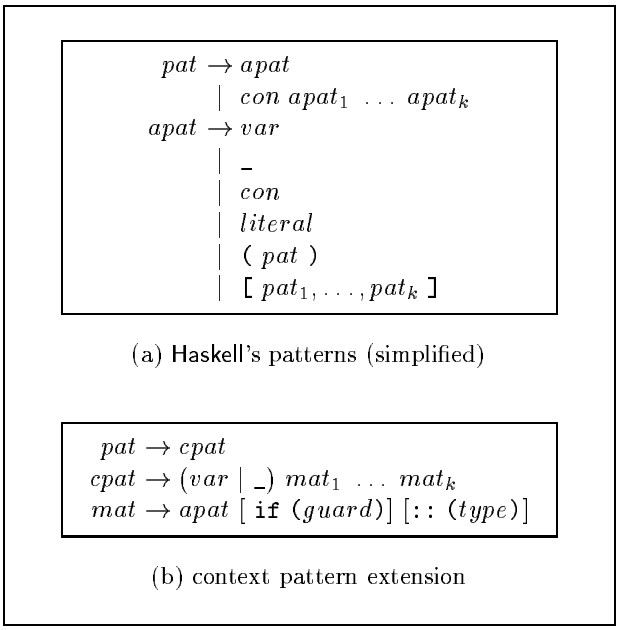


Fig. 2. Extended Haskell patterns

Type checking of patterns is done by the inference system in Figure 3, which yields sentences of the form

$$A \vdash pat \Rightarrow (A', \tau)$$

where A is an environment for variables and constructors, pat is a pattern, A' is the environment for the variables bound by pat, and τ is the type of pat. The combination of environments $A \uplus A'$ is defined iff A and A' are identical on shared entries. For the rule MAT we assume that $guard = \texttt{True}$ if no guard is present and similarly $type = \texttt{a}$. To obtain a simpler system, the rules APPL, CP1, and CP2 could be merged into one.

Axioms:

$$\frac{}{A \vdash var \Rightarrow ([var/\tau], \tau)}\ \text{VAR} \qquad \frac{}{A \vdash \_ \Rightarrow ([], \tau)}\ \text{WILD}$$

Standard Rules:

$$\frac{A(con) \succ \tau}{A \vdash con \Rightarrow ([], \tau)}\ \text{CON} \qquad \frac{A(literal) \succ \tau}{A \vdash literal \Rightarrow ([], \tau)}\ \text{LIT}$$

$$\frac{A \vdash pat_i \Rightarrow (A_i, \tau)\ (1 \le i \le n)}{A \vdash \texttt{[}\ pat_1, \ldots, pat_k\ \texttt{]} \Rightarrow (A_1 \uplus \ldots \uplus A_n, \texttt{[}\tau\texttt{]})}\ \text{LIST}$$

$$\frac{A(con) \succ \tau_1 \to \cdots \to \tau_n \to \tau \qquad A \vdash apat_i \Rightarrow (A_i, \tau_i)\ (1 \le i \le n)}{A \vdash con\ apat_1\ \ldots\ apat_k \Rightarrow (A_1 \uplus \ldots \uplus A_n, \tau)}\ \text{APPL}$$

Context Pattern Rules:

$$\frac{A(var) \succ \tau_1 \to \cdots \to \tau_n \to \tau \qquad A \vdash mat_i \Rightarrow (A_i, \tau_i)\ (1 \le i \le n)}{A \vdash var\ mat_1\ \ldots\ mat_k \Rightarrow (A_1 \uplus \ldots \uplus A_n, \tau)}\ \text{CP1}$$

$$\frac{A \vdash mat_i \Rightarrow (A_i, \tau_i)\ (1 \le i \le n)}{A \vdash \_\ mat_1\ \ldots\ mat_k \Rightarrow (A_1 \uplus \ldots \uplus A_n, \tau)}\ \text{CP2}$$

$$\frac{A \vdash apat \Rightarrow (A', \tau) \qquad A \vdash guard \Rightarrow \texttt{Bool} \qquad \tau \succ type}{A \vdash apat\ \texttt{if}\ guard\ \texttt{::}\ type \Rightarrow (A', type)}\ \text{MAT}$$

Fig. 3. Inference Rules for Patterns

Besides type correctness, there are two additional context–sensitive restrictions for Haskell's patterns:

CS1: All patterns must be linear, i.e. no repeated variable
CS2: The arity of a constructor must match the number of sub–patterns associated with it, i.e. no partially applied constructors

However, not all context patter which fulfill these conditions are to be considered well–formed: If there is no value which can possibly match the context pattern, then we consider this context pattern to be malformed. Consider the following (malformed) function definition

```
foo :: [a] -> [a]
foo (c (x:xs) (y:ys)) = exp
```

Here, we try to find two non–overlapping lists matching `(x:xs)` and `(y:ys)` with a list. However, the matching can never succeed: Obviously, a list cannot contain two non–overlapping sublists.

To avoid such malformed context pattern, we require that a context pattern $var\ mat_1\ \ldots\ mat_k$ satisfies the following context–sensitive condition:

CS3: If the context variable var has type $\tau_1 \to \ldots \to \tau_n \to \tau$, then there must exist a value v of type τ and values v_i of type τ_i such that all v_i are independent

subexpressions of v and occur in the sequence $v_1, \ldots, v_n$ in a top–down, left–to–right traversal of v.

To check this condition, we have to test if it is possible to construct such a value v. This checking is done by the inference system in Figure 4. For a type expression τ build using type variables *tyvar*, type constructors *tycon*, and function types, the sentences have the form

$$A \vdash \tau \to (\tau_1, \ldots, \tau_k)$$

where A is an environment for (data) constructors and $(\tau_1, \ldots, \tau_k)$ is a list of types which are types of independent subvalues of a value of type τ and occur in a top–down, left–to–right traversal. If we can derive the sentence $A \vdash \tau \to (\tau_1, \ldots, \tau_n, \tau_{n+1}, \ldots, \tau_{n+l})$ for a context variable of type $\tau_1 \to \ldots \to \tau_n \to \tau$ then this is equivalent to the admissibility of the context pattern according to condition *CS3*.

$$\frac{}{A \vdash \tau \to (\tau)}\ \text{REF}$$

$$\frac{A(con) = \tau_1 \to \cdots \to \tau_n \to \tau}{A \vdash \tau \to (\tau_1, \ldots, \tau_n)}\ \text{TYCON}$$

$$\frac{A \vdash \tau \to (\tau_1, \ldots, \tau_n) \quad A \vdash \tau_i \to (\tau_{i,1}, \ldots, \tau_{i,m_i})\ (1 \le i \le n)}{A \vdash \tau \to (\tau_{1,1}, \ldots, \tau_{1,m_1}, \ldots, \tau_{n,1}, \ldots, \tau_{n,m_n})}\ \text{COMB}$$

Fig. 4. Inference Rules for Condition *CS3*

Type variables and function types are solely handled by the axiom REF. Type variables are not expanded which means that no bindings can occur. Functional types are not decomposed, since no search can go into the components either. For an algebraic types with k constructors, we have $k+1$ possibilities: use rule TYCON for each constructor and use rule REF. Finally, rule COMB is to combine the results of adjacent components of constructors. An implementation of the inference system is described in [MT97].

Example: *We use the context pattern* `(c x y)` *with type* `[a]` *and the environment* $A = [$`Cons/a` $\to$ `[a]` $\to$ `[a]`, `Nil/[a]`$]$.

$$\frac{\dfrac{\text{A(Cons)=a}\to\text{[a]}\to\text{[a]}}{\text{A} \vdash \texttt{[a]} \to (\texttt{a},\texttt{[a]})} \qquad \dfrac{}{\text{A} \vdash \texttt{a} \to (\texttt{a})} \qquad \dfrac{\text{A(Cons)} = \texttt{a}\to\texttt{[a]}\to\texttt{[a]}}{\text{A} \vdash \texttt{[a]} \to (\texttt{a},\texttt{[a]})}}{\text{A} \vdash \texttt{[a]} \to (\texttt{a},\texttt{a},\texttt{[a]})}$$

Hence, the variable `x` *must have type* `a`*. Possible types for the variable* `y` *are* `[a]` *or* `a`*.*

Note that in contrast to the description in [Moh97], the check of condition *CS3* does *not* introduce new bindings of type variables. Since the construction of a value according to *CS3* is non–deterministic, the introduction of new bindings would also be non–deterministic: Potential type errors caused by the choice of bindings are completely inexplicable to the programmer. Therefore, we consider context–patterns where additional choices would have to be made as *underspecified*. In such cases additional type information must be provided by the programmer, using a type signature for the function or explicit types at the patterns.

4 Translation into Haskell

In this section we define the semantics of context patterns in terms of a translation into a Haskell program without context patterns. The idea is to implement context patterns by functions that perform the recursive search. The actual matching is done by applications of these functions.

All pattern matching constructs may appear in several places: lambda abstractions, function definitions, pattern bindings, list comprehensions, and `case` expressions. However, the first four of these can be translated into `case` expressions, so we only consider patterns in `case` expressions. Furthermore, we follow [HPW92] and assume that we have a context pattern in a *simple* `case` *expression*:

$$\texttt{case } e_0 \texttt{ \{}c\ p_1\ \texttt{if}\ g_1\ \ldots\ p_n\ \texttt{if}\ g_n\ \texttt{->}\ e\ \texttt{;\ \_ ->}\ e'\texttt{\}}$$

For the top–down left–to–right traversal of a value e_0, we have to visit every sub–expression of e_0 which may contain one of the patterns p_i. Assume that $t_0, \ldots, t_m$ are the types of all sub–expressions of e_0 such that t_0 is the type of e_0 and t_{p_i} is the type of the pattern p_i $(1 \leq i \leq n)$. To traverse all of e_0, we provide functions

$$chk_{t_j}\ \texttt{::}\ t_{args}\ \texttt{->}\ t_j\ \texttt{->}\ t_{res-j}$$

for each t_j. Each of these functions traverses one type of sub–expression. At top level, we use chk_{t_0} to perform the complete search. Hence, a first approximation of the top–level structure of the translation looks like this:

$$\begin{array}{l}\texttt{case } e_0 \texttt{ of \{}c\ p_1\ \texttt{if}\ g_1\ \ldots\ p_n\ \texttt{if}\ g_n\ \texttt{->}\ e\ \texttt{;\ \_ ->}\ e'\texttt{\}}\\ \rightsquigarrow \texttt{let \{ } \langle chk_{t_0}\text{–}decl\rangle\ \texttt{;}\ \ldots\ \texttt{;}\ \langle chk_{t_m}\text{–}decl\rangle\ \texttt{\}}\\ \quad\texttt{in case (}\ chk_{t_0}\ \langle args\rangle\ e_0\ \texttt{) of \{}\ \ldots\ \texttt{\}}\end{array}$$

Example: *In this section, we use the following program as running example. It implements a function* `rplc`*, which is a generalisation of the function* `rplc_a` *from Section 2. Here, the application* `(rplc x y l)` *yields a list where all occurrences of* `x` *in* `l` *are replaced by* `y`*.*

```
rplc :: Eq a => a -> a -> [a] -> [a]
rplc x y (c z if (z==x)) = c_e (rplc x y) y
rplc x y l = l
```

For this example, we have only one pattern $p_1 =$ `z` *and hence* $t_0 =$ `[a]` *and* $t_1 =$ `a`*. Therefore, the top–level structure of the translation is*

```
rplc x y l = let chk0 ... x0 = ...
                 chk1 ... x0 = ...
             in case (chk0 ... l) of ...
```

To fill out the dots in this scheme we have to consider which additional arguments t_{args} and which result t_{res-j} are to be provided:

(a) The functions chk_{t_j} must have information on which pattern p_i is to be searched next.
(b) The result must contain information on whether the match was successful.
(c) The result must contain bindings for the variables in the pattern p_i.
(d) The result must contain a binding for the context c and the extended context `c_e`. It is sufficient to get a binding for `c_e` since we have $c =$ (`c_e` id).

We can solve (a) and (b) by using the number of the pattern which is to matched next as component of both t_{args} and t_{res-j}: The match was successful iff the number returned by chk_{t_0} on the top–level is $n+1$.

To solve (c), we provide the complete list of variables occurring in the patterns $p_1, \ldots, p_n$ as argument and result for each function chk_{t_j}. Please note in this context that we have a function for *each subtype* of the argument and *not* a function for each pattern. Therefore it is possible that a function chk_{t_j} handles more than one pattern.

Condition (d) requires a slightly different approach: We cannot pass the binding for the context variable in the same way we passed the other bindings. The computation of the context function must be performed bottom–up because it depends on the environment of a sub–expression. Hence, each chk_{t_j} has one more result component, the *local extended context*. Such an object is a function taking a function of type $t_0 \to t_0$ and one argument for each pattern p_i and yields a result of type t_j. The local context produced by the top–level evaluation of chk_{t_0} is the binding for the context variable.

Let t_{pxs} be a tuple of the types of all variables in the pattern p_i. We can now give the complete type of chk_{t_j}

$$chk_{t_j}\texttt{::(Int,}t_{pxs}\texttt{) -> }t_j\texttt{ -> (Int,}t_{pxs}\texttt{,(}t_0\texttt{->}t_0\texttt{)->}t_1\texttt{->...->}t_n\texttt{->}t_j\texttt{)}$$

To initialise the arguments for bindings which are not yet found we use the undefined value of arbitrary type: `undefined = error ""` as defined in the prelude. The complete top–level structure of the translation now looks like this (we abbreviated `undefined` by $\perp$):

```
case e0 of {c p1 if g1 ... pn if gn -> e ; _ -> e'}
~> let { ⟨chk_t0–decl⟩ ; ... ; ⟨chk_tm–decl⟩ }
   in case (chk_t0 (1,⊥,...,⊥) e0) of {
           (n+1,x_1,1,...,x_n,kn,c_e) -> let { c = c_e id } in e;
           _ -> e' }
```

where $x_{i,1}, \ldots, x_{i,k_i}$ are the variables in p_i.

Example: *(continued) For* `rplc` *we have only one variable* `z` *in the pattern* p_1*:*

```
rplc x y l = let chk0 ... x0 = ...
                 chk1 ... x0 = ...
             in case (chk0 (1,undefined) l) of
                (2,z,c_e) -> let c = c_e id
                             in c_e (rplc x y) y
                _         -> l
```

The implementation of chk_{t_j} checks the number of the next pattern and handles each of the cases. Let n^i be new variables, $\bar{y}^i := y^i_{1,1}, \ldots, y^i_{n,k_n}$ be vectors of unused variables, one variable for each variable in the patterns, and $\bar{z} := z_1, \ldots, z_n$ be a vector of variables for the arguments of a context function:

$$\begin{aligned}\langle chk_{t_j}\text{–}decl\rangle \rightsquigarrow\ & chk_{t_j}\ (n^0,\bar{y}^0)\ x\ \texttt{= case}\ n^0\ \texttt{of 1 ->}\ \langle chk_{t_j,1}\rangle \\ & \ldots \\ & n\ \texttt{->}\ \langle chk_{t_j,n}\rangle \\ & n+1\ \texttt{->}\ \langle rchk_{t_j}\rangle\end{aligned}$$

If all patterns are found (the last case) then what remains to be done is the search for all parts of type t_0. The right hand side $\langle rchk_{t_j}\rangle$ is essentially the same as

$\langle chk_{t_j,i} \rangle$ would be for a pattern $p_i = x$ of type t_0 except that the number of patterns found is not increased and the functional parameter of the context is applied to the parts. We suspend the explanation of $\langle rchk_{t_j} \rangle$ until we have finished $\langle chk_{t_j,i} \rangle$.

For the realisation of the right hand side $\langle chk_{t_j,i} \rangle$ which checks for pattern p_i, we distinguish whether p_i can be found in a value of type t_j. Using the inference system from the last section, we can formalise this by $A \vdash t_j \to (\ldots, t_{p_i}, \ldots)$.

If it cannot be found, the implementation of $\langle chk_{t_j,i} \rangle$ is trivial, because there is no need for recursive search. Doing so would only violate the laziness, because it would force the evaluation. All we have to do is to create a trivial context. Note that in contrast to the case that all patterns are found, we do not search for part of type t_0 inside this component: The extended context applies its argument to the unvisited parts only.

However, if the pattern can occur is this sub–expression, $\langle chk_{t_j,i} \rangle$ proceeds in two steps:

1. It checks whether the pattern can occur directly (done by $\langle p_i\text{–}chk_{t_j} \rangle$).
2. It performs a recursive search in the sub–expressions (done by $\langle r_{t_j}\text{–}chk \rangle$).

Hence, we can define $\langle chk_{t_j,i} \rangle$ in the following way:

$$\langle chk_{t_j,i} \rangle \rightsquigarrow \begin{cases} \texttt{case}\ x^0\ \texttt{of}\ \{ & \\ \quad \langle p_i\text{–}chk_{t_j} \rangle & \\ \quad \langle r_{t_j}\text{–}chk \rangle & \\ \quad \_ \to (i, \bar{y}^0, \backslash f\texttt{->}\backslash \bar{z}\texttt{->}x^0) & \\ \} & \text{if } A \vdash t_j \to (\ldots, t_{p_i}, \ldots) \\ & \\ (i, \bar{y}^0, \backslash f\texttt{->}\backslash \bar{z}\texttt{->}x^0) & \text{otherwise} \end{cases}$$

If t_j is the type of p_i then the pattern can occur directly; Otherwise, there is nothing to do in $\langle p_i\text{–}chk_{t_j} \rangle$. Hence the code $\langle p_i\text{–}chk_{t_j} \rangle$ is defined as

$$\langle p_i\text{–}chk_{t_j} \rangle \rightsquigarrow \begin{cases} p_i \mid g_i \texttt{->} (\ i+1, & \text{if } t_j = t_i \\ \quad y^0_{1,1}, \ldots, y^0_{i-1,k_{i-1}}, & \\ \quad x_{i,1}, \ldots, x_{i,k_i}, & \\ \quad y^0_{i+1,1}, \ldots, y^0_{n,k_n}, & \\ \quad \backslash f\texttt{->}\backslash \bar{z}\texttt{->}z_i\ )\texttt{;} & \\ \varepsilon & \text{otherwise} \end{cases}$$

The value consist of the following entries: the number of the next pattern $(i + 1)$, updated bindings for variables in the patterns, and an extended context function returning the $i - th$ argument ($\backslash f\texttt{->}\backslash \bar{z}\texttt{->}z_i$). The updated bindings consists of three groups: the unchanged values of the variables in the preceding patterns ($y^0_{1,1}, \ldots, y^0_{i-1,k_{i-1}}$), the values of the variables in the current pattern ($x_{i,1}, \ldots, x_{i,k_i}$), and the unchanged values of the variables in the succeeding patterns ($y^0_{i+1,1}, \ldots, y^0_{n,k_n}$) (which are all equal to $\perp$). Note that each value in the vector is updated only once, because they are indexed by the number of the current pattern, which strictly increases. The construction of the context function is correct because by matching pattern p_i in the expression this sub–expression is "chopped" away.

The additional context pattern guard following the keyword `if` is implemented by the normal pattern match guard.

Example: *(continued) For the* `rplc` *example, the pattern* $p_1 =$ `z` *can occur directly in expressions of type* $t_1 =$ `a`*, but not recursively. Hence, the complete definition of* `chk1` *is:*

```
chk1 (n0,xb0) x0 = case n0 of
                     1 -> case x0 of
                          z | z==x -> (2,z,\f->\z->z)
                          _        -> (1,xb0,\f->\z->x0)
                     2 -> (2,xb0,\f->\z->x0)
```

What remains is the implementation of the recursive search, which is performed if the pattern cannot match directly, or does not match. We have to know all constructors of type t_j to handle all possible values. Let $K_{t_j,1}, \ldots, K_{t_j,n_j}$ be all constructors of type t_j and let $a_{j,i}$ be the arity of constructor $K_{t_j,i}$. The expressions $\langle r_{t_j}\text{–}chk\rangle$ which perform the recursive search have the following form (with w_i new variables):

$$\begin{array}{rcl}\langle r_{t_j}\text{–}chk\rangle & \rightsquigarrow & K_{t_j,1}\ w_1\ \ldots\ w_{a_{j,1}}\ \texttt{->}\ \langle chkrek_{t_j,1}\rangle\ \texttt{;} \\ & & \ldots \\ & & K_{t_j,n_j}\ w_1\ \ldots\ w_{a_{j,n_j}}\ \texttt{->}\ \langle chkrek_{t_j,n_j}\rangle\texttt{;}\end{array}$$

All values of type t_j will match exactly one of these cases. To define the right hand side $\langle chkrek_{t_j,k}\rangle$ we abbreviate $K := K_{t_j,k}$ and $a := a_{j,k}$. This expression must traverse all sub–expression w_l and create new context functions from the results. If $a = 0$ then there is nothing to. Otherwise, all sub–expressions are traversed from left to right and the resulting context functions are combined by using the constructor K. Let $t_{l_1}, \ldots, t_{l_a}$ be the argument types of constructor K, and let $c\_\texttt{e}^i, n^i, x$ be new variables. We define:

$$\langle chkrek_{t_j,k}\rangle \rightsquigarrow \begin{cases} (n^0,\bar{y}^0,\texttt{\textbackslash}f\texttt{->\textbackslash}\bar{z}\texttt{->}x^0) & \text{if } a = 0 \\ \begin{array}{l}\texttt{let } (n^1,\bar{y}^1,c\_\texttt{e}^1) \texttt{ = } chk_{t_{l_1}}\ (n^0,\bar{y}^0)\ w_1 \\ \quad \ldots \\ \quad (n^a,\bar{y}^a,c\_\texttt{e}^a) \texttt{ = } chk_{t_{l_a}}\ (n^{a-1},\bar{y}^{a-1})\ w_a \\ \texttt{in } (n^a,\bar{y}^a,\texttt{\textbackslash}f\texttt{->\textbackslash}\bar{z}\ \texttt{->}\ K\ (c\_\texttt{e}^1\ f\ \bar{z})\ \ldots\ (c\_\texttt{e}^a\ f\ \bar{z}))\end{array} & \text{otherwise} \end{cases}$$

Finally, we have to define $\langle rchk_{t_j}\rangle$, which performs the search for all unvisited parts of type t_0. This is done as soon as all pattern have matched and here the functional parameter for the extended context is used. We define

$$\langle rchk_{t_j}\rangle \rightsquigarrow \begin{cases} (n+1,\bar{y}^0,\texttt{\textbackslash}f\texttt{->\textbackslash}\bar{z}\texttt{->}(f\ x^0)) & \text{if } t_j = t_0 \\ \langle r_{t_j}\text{–}chk\rangle & \text{if } t_j \neq t_0,\ A \vdash t_j \to (\ldots,t_0,\ldots) \\ (n+1,\bar{y}^0,\texttt{\textbackslash}f\texttt{->\textbackslash}\bar{z}\texttt{->}x^0) & \text{otherwise} \end{cases}$$

Example: *(continued) For* `chk0`*, the function which checks sub–expressions of type* `[a]` *in the* `rplc` *program, we obtain the following definition:*

```
chk0 (n0,xb0) x0 = case n0 of
   1 -> case x0 of
        []      -> (n0,xb0,\f->\z->x0)
        (w1:w2) -> let (n1,xb1,c1)=chk1 (n0,xb0) w1
                       (n2,xb2,c2)=chk0 (n1,xb1) w2
                   in (n2,xb2,\f->\z->((c1 f z):(c2 f z)))
        _       -> (1,xb0,\bsf->\bsz->x0)
   2 -> (2,xb0,\f->\z->(f x0))
```

In summary, the complete program for `rplc` *is*

```
rplc x y l = let
  chk0 (n0,xb0) x0 = case n0 of
    1 -> case x0 of
         []     -> (n0,xb0,\f->\z->x0)
         (w1:w2) -> let (n1,xb1,c1)=chk1 (n0,xb0) w1
                        (n2,xb2,c2)=chk0 (n1,xb1) w2
                    in (n2,xb2,\f->\z->((c1 f z):(c2 f z)))
         _       -> (1,xb0,\bsf->\bsz->x0)
    2 -> (2,xb0,\f->\z->(f x0))
  chk1 (n0,xb0) x0 = case n0 of
                     1 -> case x0 of
                          z | z==x -> (2,z,\f->\z->z)
                          _        -> (1,xb0,\f->\z->x0)
                     2 -> (2,xb0,\f->\z->x0)
 in case (chk0 (1,undefined) l) of
    (2,z,c_e) -> let c = c_e id
                  in c_e (rplc x y) y
    _       -> l
```

5 Related Work

Patterns beyond the scope of the Haskell pattern have been studied in [Hec88, Fer90, Wil90] in the context of TrafoLa, a functional languages for program transformations. Their *insertion patterns* allow an effect similar to our context patterns, but instead of modelling a context as a function they introduce special *hole constructors* @n to denote the position where a match was cut from the context. Additionally, they introduce several other special purpose patterns for lists, and allow non–linear patterns, which may interfere with lazy evaluation [Pey87, p. 65]. Pattern matching usually results in a list of solutions.

An even more general approach was taken in [HL78] where *second–order schemes* are used to describe transformation rules. These allow the specification and selection of arbitrary subtrees but are not integrated in a functional language.

The language Refal [Tur86] which is used a the basis for supercompilation has strings as data structure and allows patterns like $\mathtt{s}_1\ \mathtt{e}_x\ \mathtt{s}_1$ where $\mathtt{s}_1$ is a string and $\mathtt{e}_x$ is a character.

In [Que90] patterns with *segment assignments* and their compilation are studied in the context of Lisp. The segments allow the access to parts of a matched list, e.g. the pattern `(?x ??y ?x)` matches all lists which start and end with `x`. The inner part of the list can be accessed via `y`.

A different extension of pattern matching is to use *unfree* data types, where the matching or a pattern is done by evaluation of a used–defined function. Different approaches based on this idea are Miranda's *laws*, Wadler's *views* [Wad87], and Erwig's *active patterns* [Erw97].

Another root of our work can be seen in *higher–order unification* [Hue75, SG89, DJ95]. The general approach is to synthesise λ–terms in order to find bindings for free function variable in applications, such that equations β–reduce to equal terms. In general, this problem is undecidable [Gol81]. However, for certain subclasses of generated λ–terms and equations, this problem becomes decidable [Pre94]. Especially in our case, where only the pattern can contain unbound variables, i.e. unification becomes matching, the problem is decidable [Hue75].

The representation of context by functions are related to [Hug86], where lists of type `[a]` are represented by a function of type `[a]->[a]`. Given a list `l`, the representation is obtained by `append l`. In our setting such functions can occur as a special case, where the "hole" is the rest of the list.

6 Conclusions and Future Work

In [Moh97] we have presented an extension of pattern matching called context patterns, which allow the matching of regions not adjacent to the root and their corresponding contexts as functional bindings. Typical examples of functions using this increased expressive power are functions which search and transform data structures. For special cases of transformations, however, this approach caused a superfluous repetition of the recursive search.

In this paper, we have introduced a new construct called *extended context*, which allows the definition of transformational functions without this additional overhead.

Furthermore, we have formally defined the static semantics of context patterns by an inference system and explained the translation into Haskell in detail.

As future research we plan to gain more experience in the use of context patterns and to validate the usefulness of context patterns for transformations. Therefore, we want to re–implement (parts of) the Simplifier, which is a component of the Glasgow Haskell Compiler performing simple performance–enhancing source code transformations.

To allow even finer access to the parts of the context further extensions of context patterns may be necessary.

Another possible future work can be based on the observation that a context function c and an associated extended context function c_e differ only in the presence of an argument, namely the functional argument: We have $c = (c\_\mathsf{e}\ id)$. The distinction between these two context functions could be avoided if the type system would support function types with *optional argument and default values*, similar to imperative languages like C++ or Ada. For our case, we would have only one context function, whose first parameter is an optional argument with default value id.

We have integrated context patterns in the Glasgow Haskell Compiler, based on version 2.01. The implementation is described in more detail in an accompanying paper [MT97]. However, the extension presented in this paper are not yet included. The source code can be obtained from the URL http://www-i2.informatik.rwth-aachen.de/~markusm/CP/.

References

[Bar85] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and The Foundations of Mathematics*. North–Holland, 1985.

[DJ95] D. J. Dougherty and P. Johann. A Combinatory Approach to Higher-Order E-Unification. *Theoretical Computer Science*, 139(1–2):207–242, March 1995.

[DM90] P. Deransart and J. Małuszyński, editors. *Proceedings of the 2nd International Workshop on Programming Language Implementation and Logic Programming (PLILP)*, number 456 in Lecture Notes in Computer Science. Springer–Verlag, 1990.

[Erw97] M. Erwig. Active Patterms. In Kluge et.al. [Klu97], pages 21—40.

[Fer90] C. Ferdinand. Pattern Matching in a Functional Transformational Language using Treeparsing. In Deransart and Małuszyński [DM90], pages 358–371.

[Gol81] W. D. Goldfarb. The Undecidability of the Second-Order Unification Problem. *Theoretical Computer Science*, 13(2):225–230, February 1981.

[Hec88] R. Heckmann. A Functional Language for the Specification of Complex Tree Transformations. In H. Ganzinger, editor, *Proceedings of the 2nd European Symposium on Programming (ESOP)*, number 300 in Lecture Notes in Computer Science, pages 175–190. Springer–Verlag, 1988.

[HL78] G. Huet and B. Lang. Proving and applying Program Transformations Expressed with Second Order Patterns. *Acta Informatica*, 11:31–55, 1978.

[HPW92] P. Hudak, S. L. Peyton Jones, and P. Wadler *et. al.* Report on the Programming Language Haskell — A Non-strict, Purely Functional Language. Research Report 1.2, Department of Computer Science and Department of Computing Science, March 1992.

[Hue75] G. Huet. A Unification Algorithm for Typed λ-Calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Hug86] R. J. M. Huges. A Novel Representation of Lists and Its Apllication to the Function "reverse". *Information Processing Letters*, 22(3):141–144, March 1986.

[Klu97] W. Kluge et.al., editor. *Selected Papers of the 8th International Workshop on Implementation of Functional Languages (IFL)*, number 1268 in Lecture Notes in Computer Science. Springer–Verlag, 1997.

[Moh97] M. Mohnen. Context Patterns in Haskell. In Kluge et.al. [Klu97], pages 41–58.

[MT97] M. Mohnen and S. Tobies. Implementing Context Patterns in the Glasgow Haskell Compiler. Technical Report AIB-97-04, RWTH Aachen, 1997. to be published.

[Pey87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[Pre94] C. Prehofer. Decidable Higher-order Unification Problems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction (CADE)*, number 814 in Lecture Notes in Computer Science, pages 635–649. Springer–Verlag, 1994.

[Que90] C. Queinnec. Compilation of Non-Linear, Second Order Patterns on S-Expressions. In Deransart and Małuszyński [DM90], pages 340–357.

[SG89] W. Snyder and J. Gallier. Higher Order Unification Revisited: Complete Sets of Tranformations. *Journal of Symbolic Computation*, 8(1 & 2):101–140, 1989. Special issue on unification. Part two.

[Tur86] V. F. Turchin. Program Transformation by Supercompilation. In H. Ganzinger and N. Jones, editors, *Programs as Data Objects*, number 217 in Lecture Notes in Computer Science, pages 257—281. Springer–Verlag, 1986.

[Wad87] P. Wadler. Views: A Way for Pattern–matching to Cohabit with Abstraction. In *Proceedings of the 14th Symposium on Principles of Programming Languages (POPL)*, pages 307–313. ACM, January 1987.

[Wil90] R. Wilhelm. Tree Transformations, Functional Languages, and Attribute Grammars. In P. Deransart and M. Jourdan, editors, *Attribute Grammars and their Applications*, number 461 in LNCS, pages 116–129. Springer–Verlag, 1990.