Context Patterns in Haskell

Markus Mohnen

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

Abstract. In modern functional languages, pattern matching is used to define functions or expressions by performing an analysis of the *structure of values*. We extend `Haskell` with a new non–local form of patterns called *context patterns*, which allow the matching of subterms without fixed distance from the root of the whole term. The semantics of context patterns is defined by transforming them to standard `Haskell` programs. Typical applications of context patterns are functions which *search* a data structure and possibly *transform* it.
This concept can easily be adopted for other languages using pattern matching like `ML` or `Clean`.

1 Introduction

Pattern matching in functional languages like `Haskell` [HF92, HPW92] is a powerful and elegant tool for the definition of functions. When a pattern matches a structure, the resulting bindings of the pattern variables yield access to substructures. Typically, definitions with patterns are used for structurally recursive function definitions.

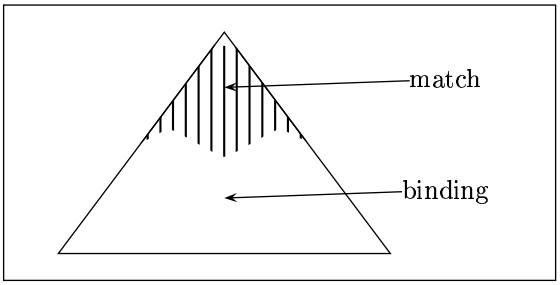


Fig. 1.: Match with standard pattern

However, the 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 Fig. 1). 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 Fig. 2).
Consider a toy example of a function

```
initlast :: [a] -> ([a],a)
```

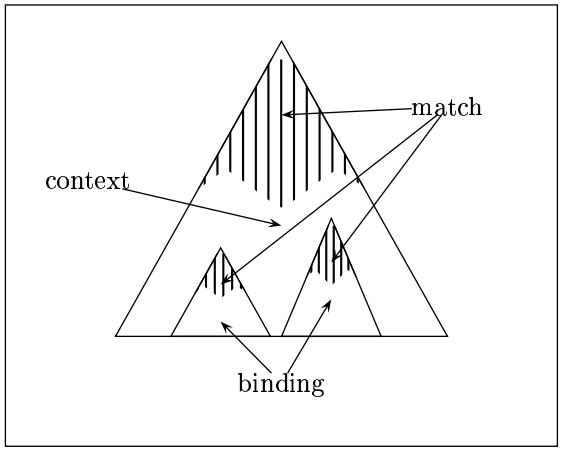


Fig. 2.: Match with context pattern

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, using the standard patterns, we are not able to express this. Instead, the recursive search must be programmed explicitly:

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

Essentially, 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" in it. The representation consists of modelling the "hole" by the function arguments, i.e. in general, f has the following form:

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

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 []      = 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:

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

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

Apart from the greater expressive power of context patterns, they provide a means to reduce the impact of any change of data structures. If a function is defined recursively for an algebraic data type, the patterns must handle all constructors. If the data type is changed during development of the program, all these functions must be changed, too. With context patterns, however, functions only need to know those constructors which are of interest for them. Therefore, not all functions must be reconsidered.

The paper is organised as follows. In Section 2 we review some of the previous work in pattern matching and topics related to our approach. Section 3 gives a formal definition of context patterns as an extension of `Haskell`'s patterns. Some examples of context pattern programs are given in Section 4. In Section 5 we define the semantics of context patterns by translating them into `Haskell` code. Section 6 concludes.

2 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 (see 3.1).

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

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`.

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 (see `initlast`).

3 Context Patterns

The syntax of `Haskell`'s patterns is shown in Fig. 3(a) (taken from [HPW92, pp. 17–18]). For simplicity, we omit infix patterns and $n + k$ patterns. Our extension is in Fig. 3(b). In addition to the basic syntax given in the introduction, there a three extras features:

- *context wildcards* (`_`), which can be used to match contexts which are not used on the right hand side
- *guards* at the sub patterns, which allow the test of additional conditions during the recursive search
- *explicit types* at the sub patterns, which allow the restriction of possible matches

Later we discuss these facilities in more detail, and give examples for their use.

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.

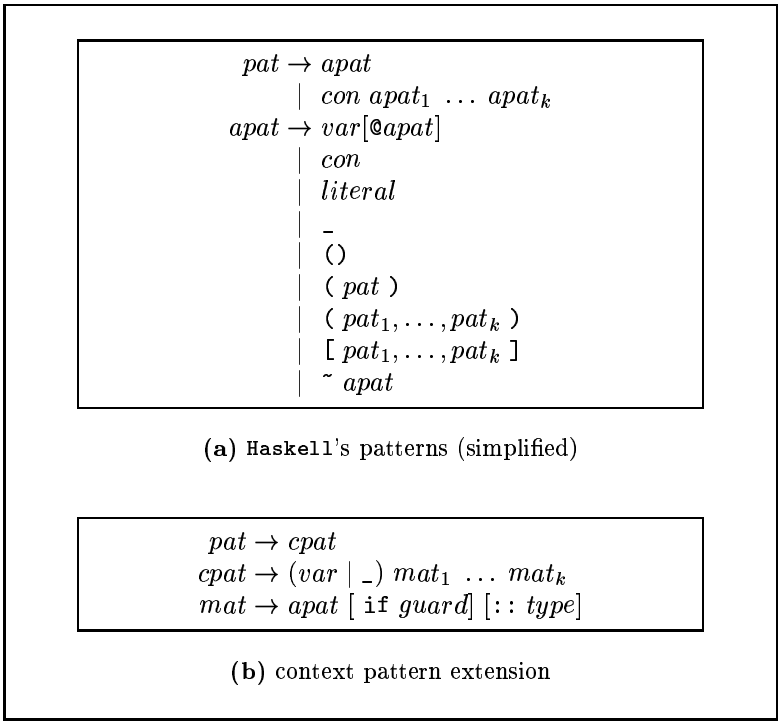
The context–sensitive conditions for `Haskell`'s patterns are

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

In addition, we require that

3. If a context variable *var* has the functional type $t_1 \to \ldots \to t_k \to t$, then there must exist a value v of type t and values v_i of type t_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.

This condition ensures that the pattern is not superfluous in the sense that it can match at least one value. If there is a t_i which can not be the type of a subexpression then no value v of type t has a subvalue v_i of type t_i and hence the context pattern can never match. The same is true if there are two t_i and



(a) `Haskell`'s patterns (simplified)

(b) context pattern extension

Fig. 3.: Extended `Haskell` patterns

t_j which are not independent. During matching, a subvalue v_j matching mat_j is not matched further. Therefore a possible match of mat_i is not checked within v_j.

To further motivate this condition, it is worthwhile to look at a few examples. Consider the following (malformed) function definition

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

This context is not admissible, because a list cannot contain two non–overlapping sublists.

The left–to–right part of condition 3 ensures that matching can be performed by traversing the value once top–down and left–to–right. For example,

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

is allowed. Matching with a list $[a_1, a_2, \ldots, a_n]$ with at least two elements results in the bindings[1]

$$\texttt{x}/a_1,\ \texttt{y}/a_2,\ \texttt{ys}/[a_3, \ldots, a_n] \text{ and } \texttt{c}/\texttt{\textbackslash z zs->(z:zs)}$$

On the other hand, switching the arguments in the pattern is not allowed:

```
bar' :: [a] -> [a]
bar' (c (x:xs) y) = exp
```

After finding a match for `(x:xs)` with a value v, there is nothing left in v to match `y`.

3.1 Uniqueness of The Solution

In general there are several possibilities to find a match for a context pattern. Consider the following function definition

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

When applying `foo` to a non–empty list we must choose deterministically which entry of the list should be bound to `x`. However, `Haskell` is a language with lazy evaluation, and therefore the match of context patterns should also be as lazy as possible. The match we choose is hence the *shortest possible*. In the above example, `x` is bound to the head of the list, `xs` to the tail, and the context `c` is bound to the identity function `\l->l`.

More precisely, the matching process traverses the value once *top–down, left–to–right*, similar to the pattern matching semantics of `Haskell` [HPW92, p. 19]. There is no `PROLOG`–like backtracking if the match fails. Hence the matching can be done in linear time in the size of the value.

An alternative possibility would be to consider *all possible* matchings, by binding variables of a context pattern to non–empty *lists* of all possible matches, if the context pattern matches at all. The advantage of this is the satisfaction of a need for *completeness of the solution*. Furthermore, in a lazy language like `Haskell` this list would also be computed lazily. If we assume that the top–down, left–to–right solution is the first element of the list, then only this element needs to be evaluated in order to check whether the context pattern matches. All other solutions can then be evaluated lazily. However, we have *not* chosen this approach for several reasons:

– *It is unclear what to do with all the solutions.* For transformational tasks, we will typically replace or modify a position and return the transformed value.

[1] Please note that the binding for `c` is a function taking *two* arguments, one for each of the *patterns* in the context pattern. A common error is to assume that `c` is a function taking *three* arguments, one argument for each *variable* in the context pattern.

Of course, we can do that for all solutions, but then we have the problem that we must recombine all these transformed values into a single result. Since the solutions need not be independent, this would be a hard task. The latter could be avoided by choosing the list of all non–overlapping solutions as the result of a context pattern match, but then we would loose the completeness again.

- *The incompleteness is already there.* Even the standard patterns of `Haskell` do not generate complete sets of solutions, due to the fact that equations are used in the order they occur in the program and are matched in a top–down, left–to–right manner.
- *The types of variables would change.* If a variable has type t in the pattern then it would have type `[`t`]` in the right hand side of the equation. This would be very confusing.

This non–deterministic approach would fit better in a integrated functional–logic language like `Curry` [HKMN95] or `Babel` [KLMNRA96], than in a purely functional one.

3.2 Additional Features

Sometimes it is necessary to restrict the possible matches of a context pattern. Suppose we use the following data structure to represent trees with a list of attributes at each node:

```
data Tree a = TNode [a] [Tree a]
```

Now consider the following context pattern:

```
fooatt (c [s]) = exp1
```

Which list in the definition of `Tree` is going to be matched here? By default, we choose the first possibility to match a list type in `Tree`, i.e. the list of attributes. Therefore the context `c` has the type `[a]->Tree a`. But suppose we want to match the last element of a non–empty successor list. Of course, we can increase the pattern accordingly:

```
foosuc (c1 (Tnode atts (c2 [s]))) = exp2
```

This definition, however, has lost the clarity of the previous one. Therefore, we allow the type of a context pattern to be restricted. Changing the above definition to

```
foosuc' (c [s]::[Tree a] ) = exp2
```

restricts the pattern `[s]` to match the tail of a list of successors.

A further extension is the possibility to move Boolean guards into the pattern. Without context patterns it is sufficient to have guards *at the end* of patterns. In the presence of context patterns, however, this is no longer satisfactory:

```
member' y (c (x:xs)) | x==y = True
member' _ _                 = False
```

At first sight, `member'` seems to be equivalent to the function `elem` from the standard prelude, i.e. it seems to implement checking for membership of an element in a list. However, this is not true. Given a list $[a_1, \ldots, a_n]$, the pattern `(c (x:xs))` matches with `x`/a_1, `xs`/$[a_2, \ldots, a_n]$, and `c`/$\lambda x.x$. After that, the guard `x==y` is checked, i.e. a_1 is compared with `y`. If it is not equal, this rule fails and the second rule is selected, yielding `False` as result.

The problem is that the guard is checked *after* the pattern was matched. If the check fails, there is no search for the *next* possible match of the pattern. In order to overcome this restriction, we allow guards inside context patterns[2]:

```
member y (_ (x:xs) if x==y) = True
member _ _                  = False
```

Now the list is searched until the pattern `(x:xs)` matches *and* the guard `x==y` becomes true.

3.3 Type Inference

During type inference, each subpattern is assigned a type, constraints between this types are recorded, and this set of constraints is solved yielding a principal type for each subpattern. In addition, the context–sensitive condition for context patterns may introduce additional constraints. Recall the introductory example `initlast`. If we omit the first line

```
initlast []     = error "Empty list"
```

then the following (preliminary) types are derived for the subpatterns of the context pattern `c [x]`:

- `a` for the context pattern `c [x]`
- `[b]->a` for the context function `c`
- `b` for the variable `x`

In order to fulfil condition 3, we then unify `[b]` and `a` and therefore obtain the following types:

- `[b]` for the context pattern `c [x]`

[2] In a preliminary version we used the guard symbol `|` instead of the keyword `if` to separate pattern and guard. But the resulting similarity between guards inside and outside context patterns was prone to cause errors.

– `[b]->[b]` for the context function `c`
– `b` for the variable `x`

3.4 Abstraction Boundaries

Assume that we have a module `M` which encapsulates an *abstract type* `Z`, and that we have another type `X` which is build by using `Z` in some way: `X = Y*Z`. Applying context patterns to values of type `X` *cannot violate the abstraction boundary.* If the functions outside `M` cannot see inside `X` then neither can context patterns. So if both `Y` and `Z` contain some type which is matched by a context pattern, then only `Y` is searched.

Of course, the same principle holds for polymorphic components: context patterns can not search inside those components.

4 Examples of Context Pattern Programs

In order to demonstrate some possible applications of context patterns, we give some example programs.

4.1 Lists As Sets

Consider the implementation of sets based on lists. A function using this representation for membership test was given at the end of Section 3. In addition, we give functions for insertion and deletion using context patterns:

```
insert x l@(_ y if y==x) = l
insert x l               = x:l
```

Here he have a context pattern inside an at–pattern, which allows the access to the whole list. Because we do not use the context function on the right hand side, we use an anonymous context of type `Eq a => a->[a]`.

```
delete x (c (y:l) if y==x) = c l
delete x l                 = l
```

The context must have type `[a]->[a]` because we must be able to *remove* an element. If `x` is the last element of the list, `l` will be bound to the empty list.

4.2 Sorting

Nested context patterns are allowed, and we can use them to implement a sort function as follows:

```
sort :: Ord a => [a] -> [a]
sort (outer (x:inner (y:zs) if y<x)) =
          sort (outer (y:inner (x:zs)))
sort zs = zs
```

If the context pattern matches `x` and `y` are of type `a` such that `x` occurs before `y` and `y<x` is true. The context `outer :: Ord a => [a]->[a]` contains everthing before `x`, the context `inner :: Ord a => [a]->[a]` everthing between `x` and `y` and the list `zs` the tail after `y`.

However, we can observe that the constructors `:` are only needed to ensure that `x` and `y` are elements. We can simplify this function in the following way:

```
sort :: Ord a => [a] -> [a]
sort (outer x (inner y if y<x)) =
          sort (outer y (inner x))
sort zs = zs
```

The context `outer` now has the type `Ord a => a -> [a] -> [a]` and still contains everthing before `y`. The tail of the list and everthing between `x` and `y` is now in the inner context `inner` of type `Ord a => a -> [a]`.

But now we have two contexts meeting directly, i.e. without a separating standard pattern and we can fuse them in the following way:

```
sort :: Ord a => [a] -> [a]
sort (c x y if y<x) = sort (c y x)
sort zs = zs
```

In this version the context `c :: Ord a => a -> a-> [a]` contains everthing but `x` and `y`. This sort of fusion is always possible when two contexts meet directly.

4.3 A Desugarer

The Glasgow Haskell Compiler `ghc` compiles `Haskell` programs by translation into an intermediate language `Core` [PS94, Pey96]. The part which performs this translation is called *desugarer*, because it removes the syntactic sugar like pattern–matching, list comprehensions, etc. One small subtask is the translation of conditionals into `case`.

Assume we represent (a subset of) `Haskell` expressions with the following data structures:

```
data Expr = EVar String         | ECon String
          | EAp Expr [Expr]     | ELam [String] Expr
          | EIf Expr Expr Expr | ECas Expr [(Pattern,Expr)]
```

```
data Pattern = PCon String [String] | PVar String
```

An expression is either a variable, a constructor, an application, a λ–abstraction, an `if`, or a `case`. Patterns are used in `case` expressions and are *flat*. Removing all conditionals can simply be done in the following way:

```
uncond :: Expr -> Expr
uncond (c (EIf ec et ef)) =
    uncond (c (ECas ec [pt,pf]))
    where pt = (PCon "True" [],et)
          pf = (PCon "False" [],ef)
uncond e = e
```

4.4 A Transformer

Another part of the `ghc` is the `Simplifier`, which transforms `Core` programs for better efficiency. One of these transformations is called *case of known constructor*. Its idea is that when the argument of a `case` is a constructor, the whole case can be removed (since all patterns are flat):

```
cokc :: Expr -> Expr
cokc (ce (ECas (EAp (ECon k1) as)
               (_ (PCon k2 vs,e) if k1==k2)))
     = cokc (ce (rplc vs as e))
cokc e = e
```

Here, we use two contexts in one pattern: the outer context `ce` is the context of the `case` expression in the whole expression, and the inner (anonymous) context `_` is the context within the list of alternatives in the `case`. The function `rplc` which replaces the constructor arguments for the pattern variables in the expression can also be defined with context patterns:

```
rplc :: [String] -> [Expr] -> Expr -> Expr
rplc [] [] e' = e'
rplc (v:vs) (e:es) e'
     = rplc vs es (rplc' v e e')

rplc' :: String -> Expr -> Expr -> Expr
rplc' v1 e (c (EVar v2) if v1==v2)
     = rplc' v1 e (c e)
rplc' v1 e e'
     = e'
```

The above examples all have the property that all occurrences of a pattern are transformed, which leads to tail–recursive functions. In these cases it is not

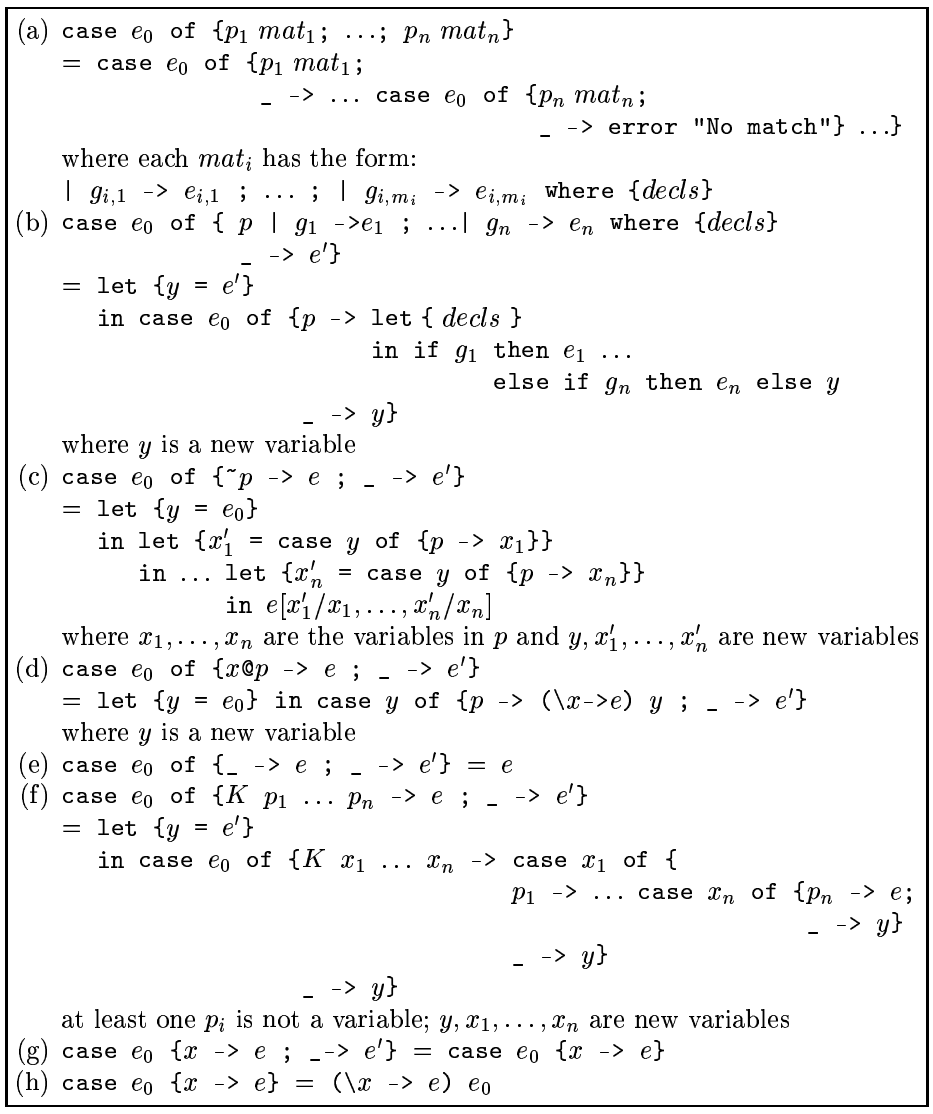


Fig. 4.: Semantics of `case` Expressions

necessary to check the complete structure again. Using this idea for an optimised implementation of context patterns leads to functions which transform the input by traversing it only once.

5 Translating Context Patterns into Haskell

All pattern matching constructs may appear in several places, i.e. 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.

In [HPW92], the semantics of `case` expressions is defined by a translation into *simple `case` expressions*:

$$\texttt{case } e_0 \texttt{ \{}K\ x_1\ \ldots\ x_n \texttt{ -> } e_1 \texttt{ ; \_ -> } e_2\texttt{\}}$$

where K is constructor (including tuple constructor) and x_i are variables. Using the rules given in Fig. 4 (taken from the semantics of `case` expressions [HPW92, p. 22]) in a left–to–right manner defines the translation. The rule (a) sequentialises `case` expressions, rule (b) removes additional guards, rules (c)–(e) remove irrefutable patterns and as–patterns, rule (f) flattens nested patterns, and rules (g) and (h) remove trivial patterns. For brevity, we omitted $n + k$ patterns.

We extend this semantics by the additional rule (cp) in Fig. 5 which remove context patterns. The computation of the context function can be omitted if it is the anonymous context. A more detailed description of the translation process can be found in an accompanying paper [MT97].

The idea is to perform a top–down left–to–right traversal of e_0. Therefore we define a function

$$chk_{t_j}\texttt{::(Int,}t_{xs}\texttt{) -> } t_j \texttt{ -> (Int,}t_1\texttt{->}\ldots\texttt{->}t_n\texttt{->}t_j\texttt{,}t_{xs}\texttt{)}$$

for each type t_j which may occur as sub–type of e_0. Each chk_{t_j} traverses a value of type t_j. By definition t_1 is the type of e_0 and hence chk_{t_1} is used to check all of e_0. The first component of the argument tuple of chk_{t_j} is the number i of the pattern p_i which is to be searched for next. The remaining arguments are bindings for all variables in the pattern found until the call to chk_{t_j}. The result tuple contains the same information after traversing the second argument and, in addition, the context created during traversal. Obviously, if the number of the next pattern is $n + 1$ all patterns were found and the match is successful.

Each chk_{t_j} is defined as a `case` on which pattern is to be matched next. If there is no more to match, all bindings and a trivial context containing the complete sub–structure are returned *without further evaluation.* Hereby, we keep the matching process as lazy as possible. For case i, we check the value for pattern p_i and guard g_i (which we assume to be `True`, if there is no guard), if this is possible with the current type t_j. If the match succeeds, we return $i + 1$ as next pattern number, an empty context, and updated bindings.

If the match is not successful or not possible at all, the value has to be examined recursively. All constructors of type t_j are matched and their arguments are traversed from left to right. At the end of the traversal, a new context is built by inserting the resulting contexts in the constructor.

Applying the transformation rule to our introductory example `initlast`

```
initlast (c [x]) = ((c []), x)
```

yields the following program:

```
case e_0 of { c p_1 if g_1 ... p_n if g_n -> e ; _ -> e' }
= let {chk_{t_1}–decl ; ... ; chk_{t_m}–decl}
  in case ( chk_{t_1} (1,⊥,...,⊥) e_0 ) of {
      (n+1, c, x_{1,1}, ..., x_{n,k_n}) -> e ;
      _ -> e' }
```

where $x_{i,1}, \ldots, x_{i,k_i}$ are the variables in p_i, $t_0, \ldots, t_m$ are the types of all sub–patterns of $c\ p_1$ `if` g_1 ... p_n `if` g_n such that t_0 is the type of the complete pattern (and e_0), t_i is the type of pattern p_i ($1 \leq i \leq n$), chk_{t_i} are new identifiers, and $\perp$ is an abbreviation for `undefined`.
We abbreviate $\bar{y}^i := y^i_{1,1}, \ldots, y^i_{n,k_n}$, $\bar{z} := z_1, \ldots, z_n$ (new variables) and define each chk_{t_j}–*decl*:

```
chk_{t_j} (n^0, ȳ^0) x = case n^0 of {
                           1 -> chk_{t_j,1} ; ... ; n -> chk_{t_j,n} ;
                           n+1 -> (n+1, (\z̄ -> x), ȳ^0)}
```

If t_i is subtype of t_j then we have to search for pattern p_i in x^0 and we define $chk_{t_j,i}$ = `case` x^0 `of {` p_i–chk_{t_j} r_{t_j}–*decl*`}`. Otherwise, pattern p_i cannot occur and hence we define $chk_{t_j,i}$ = `(`i`, (\z̄ ->` x^0`), ȳ^0)`.
If t_j is the type of p_i, then we have to check for pattern p_i and consequently p_i–chk_{t_j} is defined as

```
p_i | g_i -> (i+1, (\z̄ -> z_i), y^0_{1,1}, ..., y^0_{i-1,k_{i-1}},
                                  x_{i,1}, ..., x_{i,k_i}, ⊥, ..., ⊥);
```

Otherwise, p_i–chk_{t_j} is empty. Each r_{t_j}–*decl* performs the recursive search and has the form (w_i new variables):

```
K_{t_j,1} w_1 ... w_{a_{j,1}} -> chkrek_{t_j,1} ;
... ;
K_{t_j,n_j} w_1 ... w_{a_{j,n_j}} -> chkrek_{t_j,n_j}
```

where $K_{t_j,1}, \ldots, K_{t_j,n_j}$ are all constructors of type t_j and $a_{j,i}$ is the arity of constructor $K_{t_j,i}$. For each j,k we abbreviate $K := K_{t_j,k}$ and $a := a_{j,k}$. If $a = 0$, we define $chkrek_{t_j,k}$ = `(`n^0`, (\z̄->`x^0`), ȳ^0)`. If $a > 0$ we define:

```
chkrek_{t_j,k} = let { (n^1, c^1, ȳ^1) = chk_{l_1} (n^0, ȳ^0) w_1 } in {
                 ...
                 let { (n^a, c^a, ȳ^a) = chk_{l_a} (n^{a-1}, ȳ^{a-1}) w_a }
                 in { (n^a, (\z̄ -> K (c^1 z̄) ... (c^a z̄)), ȳ^a) } ...}
```

where $t_{l_1}, \ldots, t_{l_a}$ are the argument types of constructor K, and c^i, n^i, x are new variables.

Fig. 5.: Semantics of Context–Patterns: Rule (cp)

```
initlast = \l -> let
chk1 (n0,xb0) x0 = case n0 of
  1 -> case x0 of
       [x] -> (n0+1,(\z->z),x) ;
       []  -> (n0,(\z->x0),xb0);
       (w1:w2) ->
           let (n1,c1,xb1)=chk2 (n0,xb0) w1
               (n2,c2,xb2)=chk1 (n1,xb1) w2
           in (n2,(\z->((c1 z):(c2 z))),xb2)
       ;
  2 -> (2,(\z->z),xb0); ;
chk2 (n0,xb0) x0 = case n0 of
  1 -> (n0,(\z->x0),xb0);
  2 -> (n0,(\z->x0),xb0); ;
 in case (chk1 (1,undefined) l) of
     (2,c,x) -> (c [],x);
     _       -> undefined
```

Of course, the resulting programs can be optimised. We can distinguish three classes of possible optimisations:

1. simple well–known program transformations like *inlining* (e.g. `chk2`) and β–*reduction* (e.g. application of `chk2`) as for instance used in `ghc` [PS94, Pey96]
2. more efficient pattern matching strategies like those in [Pey87, Chapter 7] or [Thi93] can be adopted
3. completely new optimisations can be performed. If the function using a context pattern is recursive, it may be unnecessary to check the complete structure again. Using this idea on the programs given in Section 4 yields implementations which traverse each input only once.

6 Conclusion

The context patterns we have presented are a flexible and elegant extension of traditional 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/or transform data structures. Moreover, context patterns allow the definition of functions which are less affected by representation changes than usual definitions. We have presented the semantics of context patterns in terms of a translation into `Haskell`, which also gives us a first possibility for an implementation. Our next aim is to implement this translation in the Glasgow Haskell Compiler and to investigate possible optimisations. The translation and the integration is described in greater detail in an accompanying paper [MT97].

Although we have presented context patterns in the language `Haskell`, this concept can easily be adopted for other (functional) languages using pattern matching like `ML` [Mil84], `Clean` [BvELP87], or `PIZZA` [OW97].

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.

[BvELP87] T. Brus, M. van Ecklen, M. Van Leer, and M. Plasmeijer. Clean – A Language for Functional Graph Rewriting. In G. Kahn, editor, *Proccedings of the 3rd Conference on Functional Programming Languages and Computer Architecture (FPCA)*, number 274 in Lecture Notes in Computer Science, pages 364–384. Springer–Verlag, September 1987.

[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. *Proccedings of the 2nd International Workshop on Programming Language Implementation and Logic Programming (PLILP)*, number 456 in Lecture Notes in Computer Science. Springer–Verlag, 1990.

[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, *Proccedings of the 2nd International Symposium on European Symposium on Programming (ESOP)*, number 300 in Lecture Notes in Computer Science, pages 175–190. Springer–Verlag, 1988.

[HF92] P. Hudak and J. H. Fasel. A Gentle Introduction to Haskell. Technical report, Department of Computer Science, 1992.

[HKMN95] M. Hanus, H. Kuchen, and J. J. Moreno-Navarro. Curry: A Truly Functional Logic Language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.

[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.

[KLMNRA96] H. Kuchen, R. Loogen, J.J. Moreno-Navarro, and M. Rodriguez-Artalejo. The Functional Logic Language BABEL and its Implementation on a Graph Machine. *New Generation Computing*, 14:391–427, 1996.

[Mil84] R. Milner. *The standard ML core language*. Dept Computer Science, University of Edinburgh, 1984.

[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.

[OW97] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proccedings of the 24th Symposium on Principles of Programming Languages (POPL)*, pages 146–159. ACM, January 1997.

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

[Pey96] S. L. Peyton Jones. Compiling Haskell by Program Transformations: A Report from the Trenches. In H. R. Nielson, editor, *Proccedings of the 6th International Symposium on European Symposium on Programming (ESOP)*, number 1058 in LNCS, pages 18–44. Springer–Verlag, 1996.

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

[PS94] S. L. Peyton Jones and A. Santos. Compilation by Transformation in the Glasgow Haskell Compiler. In *Functional Programming, Glasgow 1994*, Workshops in Computing. 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.

[Thi93] P. Thiemann. Avoiding repeated tests in pattern matching. In P. Cousot, M. Falaschi, G. Filè, and A. Rauzy, editors, *Proccedings of the 3rd International Workshop on Static Analysis (WSA)*, number 724 in Lecture Notes in Computer Science, pages 141–152. Springer–Verlag, 1993.

[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.