Implementing Context Patterns in the Glasgow Haskell Compiler

Markus Mohnen Stephan Tobies

Lehrstuhl für Informatik II, RWTH Aachen, Germany
mohnen@informatik.rwth-aachen.de
tobies@i2.informatik.rwth-aachen.de

Abstract. Context patterns are a new non–local form of patterns, which allow the matching of subterms without fixed distance from the root of the whole term. Typical applications of context patterns are functions which *search* a data structure and possibly *transform* it, especially functions which operate on *programs as data objects*.
In this paper we describe our approach on extending the Glasgow Haskell Compiler (ghc) with context patterns.

1 Introduction

In [Moh97] we introduced *context patterns*, a new non–local form of patterns, which allow the matching of subterms without fixed distance from the root of the whole term. The main motivation for context patterns is the need for an adequate method for describing *transformations*, e.g. in optimising compilers written in Haskell [PS94, Pey96]. Such functions typically *recursively search* for an instance a subpattern in a value (aka program) and *replace* this instance. Hence, the subpattern of interest is at a non–fixed distance from the root, and the surrounding program fragment is its context.

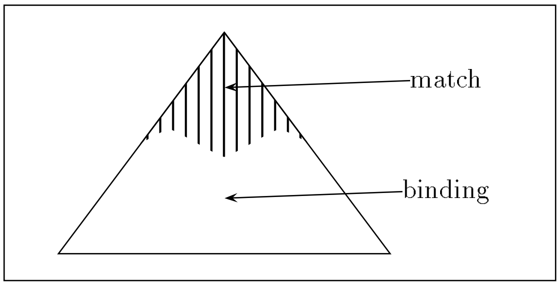

Fig. 1. Match with standard pattern

Using Haskell's standard patterns, however, this process cannot be described directly because 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). Hence, every transformation must explicitly perform the recursive search. For instance the `c2d` function in the `deforest/Core2Def.lhs` module from `ghc` version 2.01, which "translates the CoreProgram into a DefCoreProgram" consists of 31 lines of code (w/o comments) of which only 5 do the actual transformation.

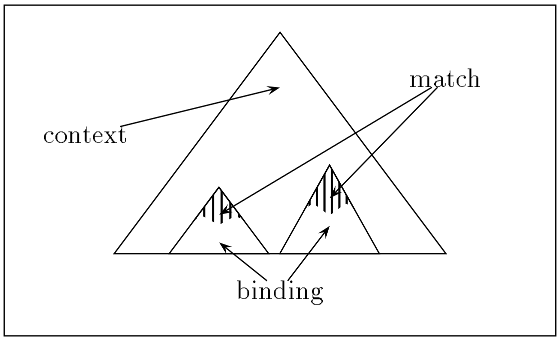

Fig. 2. Match with context pattern

In order to show a simple example of context patterns, we consider a toy 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, 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 following 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. 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:

$$[\mathtt{x}/a_n, \mathtt{c}/\backslash\mathtt{h}\text{->}(a_1\mathtt{:}\ldots(a_{n-1}\mathtt{:h})\ldots)]$$

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

In order to demonstrate how to program transformations using context patterns, we give one more example program. More examples and a detailed discussion on related work can be found in [Moh97].

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 the *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 [(Patt,Expr)]
data Patt = 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
```

The paper is organised in the following way: Section 2 gives a formal definition of context patterns as an extension of `Haskell`'s patterns. The integration in the `ghc` is discussed in Section 3. In Section 4 we give a detailed explanation of the translation of context patterns to `Haskell` code. Prospects for future work are presented in Section 5. Section 6 concludes.

2 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

`let x (y:ys) =` e_1 `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 give precedence.

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 $t_1 \rightarrow \ldots \rightarrow t_k \rightarrow t$, 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.

$$
\begin{array}{rl}
pat & \to apat \\
 & \mid\ con\ apat_1\ \ldots\ apat_k \\
apat & \to var[\texttt{@}apat] \\
 & \mid\ con \\
 & \mid\ literal \\
 & \mid\ \texttt{\_} \\
 & \mid\ \texttt{()} \\
 & \mid\ \texttt{(}\ pat\ \texttt{)} \\
 & \mid\ \texttt{(}\ pat_1, \ldots, pat_k\ \texttt{)} \\
 & \mid\ \texttt{[}\ pat_1, \ldots, pat_k\ \texttt{]} \\
 & \mid\ \texttt{\~{}}\ apat
\end{array}
$$

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

$$
\begin{array}{rl}
pat & \to cpat \\
cpat & \to (var \mid \texttt{\_})\ mat_1\ \ldots\ mat_k \\
mat & \to apat\ [\ \texttt{if}\ guard]\ [\texttt{::}\ type]
\end{array}
$$

(b) context pattern extension

Fig. 3. Extended `Haskell` patterns

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 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]

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

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`.

2.1 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:

[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.

– *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. 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.

2.2 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]) = exp_1

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]))) = exp_2

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]) = exp_2

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.

2.3 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. Our first approach ws to allow the context–sensitive condition for context patterns to 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`

To fulfil condition 3, we could unify `[b]` and `a` and therefore would obtain the following types:

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

However, but this leads to highly incomprehensible programs and error messages. Therefore, we revised this approach and reject a context pattern as underspecified if such additional unifications would be necessary.

2.4 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.

3 Integration of Context Patterns in the `ghc`

3.1 The Glasgow Haskell Compiler

The Glasgow Haskell Compiler (`ghc`) is a highly optimising compiler for `Haskell` which follows the paradigm of compiling as program transformations [PS94]: The source program is compiled into an executable form by applying a number of semantics maintaining transformations.

The major steps of this compilation process are shown in Figure 4.

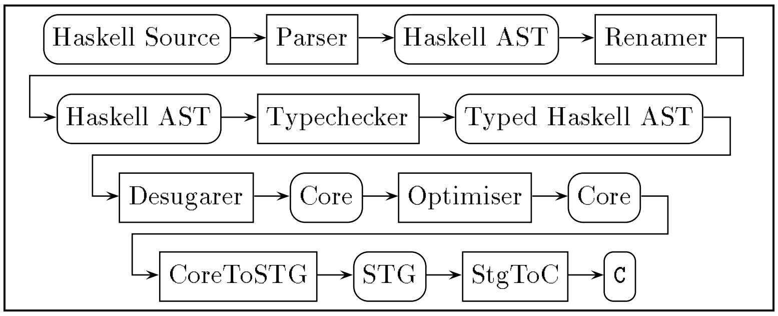

Fig. 4. The Structure of the `ghc`

To encourage the development and implementation of program optimisations for the Core intermediate language, one of the major goals of the design of the ghc was the extensibility of the optimisation phase. Consequently there exist a number of papers which deal with the details of these additions to the ghc, and there is a vast infrastructure in the compiler to help this task. Also the underlying data structures are well documented.

Unfortunately, our contribution is not of this nature. Context patterns are an addition the Haskell's syntax, which have a natural translation into Haskell code. Moreover, the translation of context pattern into Haskell essentially requires information about the types of the constructs which appear in the context pattern expression. Hence a simple pre–processor run is not sufficient to translate the context patterns into Haskell. This means that this translation has to be postponed until all necessary type information has been inferred during the compilation process. So the stage of the compiler which seems perfect for the task of the compilation of context pattern into Haskell is the desugaring stage, in which all advanced Haskell constructs are turned into Core code which is the basis for the optimisation phase of the compiler. Placing the translation there seems to be natural and enables us to make use of the optimisation also for the generated code for the context patterns.

Yet the problems which arise, when trying to implement context patterns in this manner, are manifold. The changes to the compiler are not limited to a single stage, but effect the whole frontend of the ghc, most of which is far from being well–documented. Consequently the implementation, even though the changes are very limited, took quite a lot of time, most of which was spent trying to understand what really happens in the ghc. In the remaining of this report we present these changes to the frontend of this compiler.

3.2 The Parser

Most parts of the ghc are written in Haskell, the only major exception is the parser, which is implemented using YACC and C. This is done for efficiency reasons, because at this time there are no Haskell based parsers which can compete with the speed of a YACC generated parser. In the near future this is likely to change and the C parser will be replaced by a Happy [Verweis auf irgendwas ueber Happy] generated parser. To be able to handle the context pattern syntax, the present YACC grammar had to be extended, yet the position and nature of these changes is very straightforward (at least as straightforward as a change of a 1500+ lines YACC grammar can be). Very helpful for this issue was to require the guards and type constraints to appear within brackets. Without this, the number of shift/reduce conflicts rises from 12 to 209.

```
cpat          : qvarid mats
              | WILDCARD mats
              ;
mats          : mat
              | mats mat
              ;
mat           : apat maybecpatif maybecpattype
```

```
                ;
maybecpatif   : IF OPAREN oexp CPAREN
              |
              ;
maybecpattype : DCOLON OPAREN ctype CPAREN
              |
              ;
```

Once the parser has turned the input program into its syntax tree representation, this syntax tree has to be turned into a `Haskell` data structure to be manipulated by the remaining (`Haskell` written) compiler. The `ghc` has a `C` call feature which allows to call `C` functions from within a `Haskell` program. To make the translation process as easy and safe as possible, the compiler handles this task by relying on a set of programs which do this automatically and which are generated from abstract definitions of the syntax tree which are used to generated corresponding `C` and `Haskell` data structures which can be turned into each other. Hence these definitions had to be extended to contain a construct for context patterns.

3.3 The Reader

The reader is the next stage of the compiler. It performs including of modules and scope analysis. To make the reader able to handle context pattern, only minor changes have been necessary.

3.4 The Typechecker

The most complex task while implementing context patterns, certainly have been the changes to the typechecker, because the typechecker is a very complex piece of code. Firstly let us inspect the kind on entries in the abstract syntax tree we are dealing with. Inside the typechecker, the syntax tree comes in two flavours, the *in* and the *out* flavour which stand for the syntax tree before and after the typchecking process respectively. Here are the additions to the datatype declarations for the InPat datatype.

```
data InPat name
  = WildPatIn                          -- wild card
  | VarPatIn       name                -- variable
  ...
  -- additions for context patterns
  | ContextPatIn                       -- context pattern
      (InPat name)                     -- context variable
      [CPatMatchIn name]               -- matches

data CPatMatchIn name
  = CPatMatchIn
```

```
        (InPat name)
        (Maybe (HsExpr Fake Fake name (InPat name)))
        (Maybe (PolyType name))
```

It is noteworthy that the context variable is represented as a pattern. This is necessary because a context variable can either be a normal variable or a wildcard. The matches are stored as a list with elements of type CPatMatchIn which collect the information about the different matches of the context pattern: the pattern to match, an optional guard, and an optional type constraint.

The OutPat datatype is extended accordingly to express information about the typechecked context pattern.

```
data OutPat tyvar uvar id
  = WildPat         (GenType tyvar uvar) -- wild card
  | VarPat          id                   -- variable
  ...
  -- addition for context patterns
  | ContextPat                           -- context pattern
      (OutPat tyvar uvar id)             -- context variable
      [CPatMatch tyvar uvar id]          -- matches
      (GenType tyvar uvar)               -- the type of the
                                         -- context pattern

data CPatMatch tyvar uvar id
  = CPatMatch
      (OutPat tyvar uvar id)
      (Maybe (HsExpr tyvar uvar id (OutPat tyvar uvar id)))
      (Maybe (PolyType (OutPat tyvar uvar id)))
```

Typechecking of context patterns is integrated into the normal typechecking process. To typecheck a context pattern, firstly the pattern which contains the context variable is typechecked to assign a type variable to it. This type variable, together with the location of the context pattern in the source is recorded to enforce the constraints on the type of a context pattern as they were described before and to make error reporting possible if these constraints are found to be violated. After this all matches of the context pattern are checked subsequently and the type of the context variable is modified accordingly if possible. Any errors which are found during this process are reported using the usual error reporting machinery.

After the whole module has been typechecked, but before the type variables are "zonked" (turned from mutable variables into their immutable representations), the context pattern type constraint is enforced. All necessary information has been recorded during the type checking process and is now passed to a function which checks the constraints to be valid and hence if the context pattern is admissible. As said before, we only record the type of the context function (and the source location for error reporting), but this information is sufficient to be able to check the type constraint.

If a context variable has the type $T_1 \to T_2 \to \ldots \to T_n \to T$ that means for the context pattern to be admissible there has to be an instance of type T which contains the type $T_1, \ldots, T_n$ in that order as subtypes. Hence T is checked if it has this property.

A first draft of the type constraint allowed this stage of typechecking to lead to further specialisation of types, but this led to highly incomprehensible programs and was revised, so that now in this stage no further unification of type variables is allowed to take place. If the check fails at all or would lead to further specialisation of the type T an appropriate error message is produced.

If this check succeeds the typechecker continues with its work, otherwise it exits with an error message.

3.5 The Desugarer

The next stage of the compiler is the desugarer. This is the last stage which is effected by our implementation. The desugarer turns `Haskell` constructs into Core, an intermediate language which is a slight extension of the second order λ–calculus with `let` and `case` constructs to handle common expressions and algebraic datatypes.

In the desugarer there were two possibilities to translate context pattern into Core expressions. The first was to translate context pattern into Core directly, the second the produce `Haskell` code for the context pattern and use the desugarer to translate this code into Core. We chose the second way because the translation for context patterns, which is presented in more detail in the next section, uses `Haskell` as target for the translation of context pattern. Furthermore this `Haskell` code is rather complex, using pattern matching itself, and hence it would be a tedious task to generate corresponding Code "by hand" and without support from the desugarer.

The problem which arises with the use of this method is that the desugarer expects its input `Haskell` programs to by fully annotated with their types. So we could either generate `Haskell` code and use the typechecker to annotate it with its type, or we could do this annotation directly using the type information stored with the context pattern at this stage of compilation. Although the second way seems to be complicated and the most effort is actually spent passing type information around and arranging it to annotate the generated `Haskell` code, that way had to be taken because the typechecker is unavailable in the desugaring stage of the compiler and a change in the compiler to make it possible to call the typechecker from within the desugarer would have been of such a major nature that is seemed rather impossible.

In the present implementation we follow the outlines presented in the translation of context pattern and construct an abstract `Haskell` syntax representation of the given code, which is passed to the desugarer to get the corresponding Core code which is chained into the Core code for the constructs surrounding the context pattern.

Only the pattern match which checks the value of the variable n after the call of the `check` function if the check was successful. This pattern match is generated directly in Core to make it possible to plug in the fail expression which has to be evaluated if the check fails in at the right position.

The function which performs this task is called from within the ordinary `matchUnmixedEqns` function which replaces blocks of `Haskell` patterns by their Core translations:

```
matchUnmixedEqns all_vars@(var:vars) eqns_info shadows
  | irrefutablePat first_pat
  = ...
  | isConPat first_pat
  = ...
  | isLitPat first_pat
  = ...
  | isContextPat first_pat
  = matchContextPat all_vars eqns_info shadows

  where
    first_pat     = head column_1_pats
    column_1_pats = [pat | EqnInfo (pat:_) _ <- eqns_info]
```

All further details of the implementation can be found in the sources which is written in literate style and contains a number of thoughts about the implementation and further explanations.

3.6 Features and Bugs

Let us start with the benefits and features of our implementation: The changes to the compiler are of very limited nature. Some lines here and there which allow the present machinery to deal with context patterns and two dedicated files which collect special purpose functions to be called from within the typechecker and desugarer make it feasible to carry this implementation to new versions of the `ghc` which at the moment is undergoing a fast successions of new revisions.

To compile context pattern into Core in the desugarer lets them have benefit from the vast number of optimisations which the compiler applies to Core programs. The compilation via `Haskell` code makes it possible to change the code context patterns are translated into to different (more efficient) code, or even to extend context patterns to have a different operational semantic. If the translation would produce Core code directly this certainly would be a harder task.

Now you will probably want to get to know the bugs as well. Well, there are not many (at least we have not found any). The efficiency issue is still unsolved for this version of context patterns, but that issue is tackled by a revision of the context pattern idea. What is not done, but should be, is to generate different code in case that the context variable is a wildcard. In that case we do not need to generate code which collect information about the context functions and hence the generated code could be more compact and would run more efficiently.

All these issues will be handled by a new implementation of context pattern which will include the new revision, which is to be developed for a more current version of the `ghc`.

4 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*:

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

where K is constructor (including tuple constructor) and x_i are variables. Using the rules given in Fig. 5 (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 now extend this translation by a rule (cw) (see Fig. 6) which translate context patterns. Therefore, we can assume that we have the following situation:

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

where we have the context pattern at the root and an one alternative. If the pattern p_i has no guard, we assume that g_i = `true`. This kind of situation is created by the rules from Fig. 5.

In order to describe how the translation of context patterns is done, we stepwise develop the resulting code. 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. 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. These functions traverses this type of sub–expression. Then we can use chk_{t_0} to perform the complete search. Hence, a first approximation of the top–level structure of the translation looks like this:

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

For `initlast` we have $t_0 =$ `[a]` and $t_1 =$ `a` and hence

```
initlast l = let chk0 ... x0 = ...
                 chk1 ... x0 = ...
             in case (chk0 ... l) of
                  ... -> (c [],x)
                  _   -> undefined
```

In order to fill out the dots 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 variable c

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 $n+1$.

In order 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 context*. Such an object is a function taking one argument for each pattern p_i and yields a result of type t_j. Again, 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}

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

In order 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 $\bot$):

$$
\begin{array}{l}
\texttt{case } e_0 \texttt{ of \{} c\ p_1 \texttt{ if } g_1 \ \ldots\ p_n \texttt{ if } g_n \texttt{ -> } e \texttt{ ; \_ -> } e' \texttt{\}} \\
= \texttt{let \{ } chk_{t_0}\text{–}decl \texttt{ ; } \ldots \texttt{ ; } chk_{t_m}\text{–}decl \texttt{ \}} \\
\quad \texttt{in case (} chk_{t_0} \texttt{ (1,} \bot \texttt{,} \ldots \texttt{,} \bot \texttt{) } e_0 \texttt{) of \{} \\
\qquad\qquad \texttt{(} n+1 \texttt{,} x_{1,1} \texttt{,} \ldots \texttt{,} x_{n,k_n} \texttt{,} c \texttt{) -> } e \texttt{;} \\
\qquad\qquad \texttt{\_ -> } e' \texttt{ \}}
\end{array}
$$

where $x_{i,1}, \ldots, x_{i,k_i}$ are the variables in p_i. For `initlast` we have but one such variable `x` and the above rule yields the following program fragment:

```
initlast l = let chk0 (n0,xb0) x0 = ...
                 chk1 (n0,xb0) x0 = ...
             in case (chk0 (1,undefined) l) of
                (2,x,c) -> (c [],x)
                _       -> undefined
```

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. We define chk_{t_j}–*decl* in the following way:

$$
\begin{array}{ll}
chk_{t_j}\ \texttt{(}n^0\texttt{,}\bar{y}^0\texttt{)}\ x\ \texttt{= case}\ n^0\ \texttt{of} & \\
\qquad\qquad\qquad\qquad \texttt{1 ->}\ chk_{t_j,1} & \\
\qquad\qquad\qquad\qquad \ldots & \\
\qquad\qquad\qquad\qquad n\ \texttt{->}\ chk_{t_j,n} & \\
\qquad\qquad\qquad\qquad n+1\ \texttt{->}\ \texttt{(}n+1\texttt{,}\bar{y}^0\texttt{,}\backslash\bar{z}\texttt{->}x\texttt{)} &
\end{array}
$$

If all patterns are found (the last case) then all chk_{t_j} has to do is create a trivial context, i.e. a constant function returning the argument of chk_{t_j}.

For the realisation of the right hand side $chk_{t_j,i}$ which checks for pattern p_i, we distinguish two cases: Either p_i can be found in t_j, or it cannot because the type t_{p_i} of p_i is no subtype of t_j.

In the second case the implementation of $chk_{t_j,i}$ is trivial, because there is no need for recursive search. Doing so would only violate the laziness, because it would force the evaluation. Again, we construct a trivial context.

For our running example, we now can give the complete definition of `chk1`, the function which checks sub–expression of type `a`.

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

Obviously, this function always returns immediately. But this is what we expected, because the pattern `[x]` cannot not occur inside a expression of type `a`.

On the other hand, if the pattern can occur is this sub–expression, $chk_{t_j,i}$ proceeds in two steps: First, it check whether the pattern can occur directly (done by p_i–chk_{t_j}), and secondly, it performs a recursive search in the sub–expressions (done by r_{t_j}–*decl*). Hence, we can define $chk_{t_j,i}$ in the following way:

$$
chk_{t_j,i} = \begin{cases} \texttt{case}\ x^0\ \texttt{of \{}\ p_i\text{–}chk_{t_j}\ \ r_{t_j}\text{–}decl\ \texttt{\}} & \\ \qquad\qquad\qquad\qquad \text{if } t_{p_i} \text{ is subtype of } t_j & \\ \texttt{(}i\texttt{,}\bar{y}^0\texttt{,}\backslash\bar{z}\texttt{->}x^0\texttt{)} \quad \text{otherwise} & \end{cases}
$$

The pattern can occur directly, if t_j is the type of p_i, i.e. if $t_j = t_{p_i}$. In this case p_i–chk_{t_j} is defined as

```
pᵢ | gᵢ -> ( i+1,
             y⁰₁,₁, ..., y⁰ᵢ₋₁,ₖᵢ₋₁,
             xᵢ,₁, ..., xᵢ,ₖᵢ,
             y⁰ᵢ₊₁,₁, ..., y⁰ₙ,ₖₙ,
             \z̄ -> zᵢ);
```

Here, we can implement the additional context pattern guard `if` by means of the normal guard.

The value consist of the following entries: the number of the next pattern $(i+1)$, updated bindings for variables in the patterns, and a context function returning the $i-th$ argument ($\backslash\bar{z}$ `->` z_i). The updated bindings consists of three groups: the unchanged values of the variables in the preceding patterns ($y^0_{1,1}$,...,$y^0_{i-1,k_{i-1}}$), the values of the variables in the current pattern ($x_{i,1}$,...,x_{i,k_i}), and the unchanged values of the variables in the succeeding patterns ($y^0_{i+1,1}$,...,y^0_{n,k_n}) (which are all equal to $\perp$). Note that each value in binding vector is updated only once, because they are index 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.

If the pattern cannot occur directly, p_i–chk_{t_j} is empty.

What remains is the implementation of the recursive search, which is performed if the pattern cannot match directly, or does not match. Now we must know all constructors of type t_j in order 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 r_{t_j}–$decl$ which perform the recursive search have the following form (with w_i new variables):

```
Kₜⱼ,₁ w₁ ... wₐⱼ,₁ -> chkrekₜⱼ,₁ ;
...
Kₜⱼ,ₙⱼ w₁ ... wₐⱼ,ₙⱼ -> chkrekₜⱼ,ₙⱼ
_ -> (n⁰,ȳ⁰,\f->\z̄->x⁰)
```

All values of type t_j will match exactly one of these cases. In order to define $chkrek_{t_j,k}$ 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 and we define

$chkrek_{t_j,k}$ = (n^0,$\bar{y}^0$,$\backslash\bar{z}$->x^0).

On the other hand, if $a > 0$ then 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^i, n^i, x be new variables. We define

```
chkrekₜⱼ,ₖ = let (n¹,ȳ¹,c¹) = chkₜₗ₁ (n⁰,ȳ⁰) w₁
                 ...
                 (nᵃ,ȳᵃ,cᵃ) = chkₜₗₐ (nᵃ⁻¹,ȳᵃ⁻¹) wₐ
             in (nᵃ,ȳᵃ,\z̄ -> K (c¹ z̄) ... (cᵃ z̄))
```

For `chk0`, the function which checks sub–expression of type `[a]` in the `initlast` program, we obtain the following definition:

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

The complete program for `initlast` can be found in the appendix.

The approach described above is formalised by the additional rule (cp) in Fig. 6 which removes context patterns. In [Moh97] we also give an optimised rule (cw) for anonymous context variables, where the computation of the context is removed.

5 Future Work

Our next aim is to investigate optimisations of the translation. We can distinguish three classes of possible optimisations:

1. simple well–known program transformations like *inlining* (e.g. `chk1`) and β–*reduction* (e.g. application of `chk1`) as for instance used in `ghc` [PS94, Pey96]
2. more efficient pattern matching strategies like those in [Pey87, Chapter 7] or [Thi93] may be adopted
3. completely new optimisations can be performed. For instance, if the function using a context pattern is recursive, it may be unnecessary to check the complete structure again. This occurs is the `uncond` example from the introduction. For this optimisation, however, it is necessary to compile to abstract machine code, because we cannot express the continuation of a interrupted search on the `Haskell` level.

   On other possible optimisation is the context merging described for the sort example in Section 2.

Closely related is the search for further extension. Especially when dealing with transformations, it is typically the case that all occurrences of a pattern must be replaced. For the `uncond` example in the introduction, we used tail recursion in combination with context patterns. Because the initial fragment is searched multiple times this is not efficient, and it might be possible to provide a special construct for this instead of trying to optimise it.

Another possible extension is the pruning of subexpression. For some transformations it is necessary to restrict the area in which the patterns are searched. Currently, it is not possible to express this.

```
(a) case e0 of { p1 mat1 ; ... ; pn matn }
    = case e0 of { p1 mat1;
                   _ -> ... case e0 of { pn matn;
                                         _ -> error "No match" } ... }
    where each mat_i has the form:
    | g_i,1 -> e_i,1 ; ... ; | g_i,m_i -> e_i,m_i where {decls}
(b) case e0 of { p | g1 ->e1 ; ...| gn -> en where {decls}
                 _ -> e' }
    = let {y = e'}
      in case e0 of { p -> let { decls }
                           in if g1 then e1 ...
                                 else if gn then en else y
                      _ -> y }
    where y is a new variable
(c) case e0 of { p -> e ; _ -> e' }
    = let {y = e0}
      in let {x'1 = case y of { p -> x1}}
         in ... let {x'n = case y of { p -> xn}}
                in e[x'1/x1, ..., x'n/xn]
    where x1, ..., xn are the variables in p and y, x'1, ..., x'n are new variables
(d) case e0 of { x@p -> e ; _ -> e' }
    = let { y = e0 } in case y of { p -> (\x->e) y ; _ -> e' }
    where y is a new variable
(e) case e0 of { _ -> e ; _ -> e' } = e
(f) case e0 of { K p1 ... pn -> e ; _ -> e' }
    = let { y = e' }
      in case e0 of { K x1 ... xn -> case x1 of {
                                       p1 -> ... case xn of { pn -> e;
                                                              _ -> y }
                                       _ -> y }
                      _ -> y }
    at least one p_i is not a variable; y, x1, ..., xn are new variables
(g) case e0 { x -> e ; _-> e' } = case e0 { x -> e }
(h) case e0 { x -> e } = (\x -> e) e0
```

Fig. 5. Semantics of case Expressions

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

$= \texttt{let \{}\ chk_{t_0}\text{–}decl\ \texttt{;}\ \ldots\ \texttt{;}\ chk_{t_m}\text{–}decl\ \texttt{\}}$

$\quad \texttt{in case (}\ chk_{t_0}\ \texttt{(1,}\bot\texttt{,}\ldots\texttt{,}\bot\texttt{)}\ e_0\ \texttt{) of \{}\ \texttt{(}n+1\texttt{,}x_{1,1}\texttt{,}\ldots\texttt{,}x_{n,k_n}\texttt{,}c\texttt{)}\ \texttt{->}\ e\ \texttt{;}$
$\qquad \_\ \texttt{->}\ e'\ \texttt{\}}$

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 $c\ p_1\ \texttt{if}\ g_1\ \ldots\ p_n\ \texttt{if}\ g_n$ such that t_0 is the type of the complete pattern (and e_0), t_{p_i} is the type of pattern p_i ($1 \leq i \leq n$), chk_j are new identifiers, and $\bot$ 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}–$decl$:

$$chk_{t_j}\ \texttt{(}n^0\texttt{,}\bar{y}^0\texttt{)}\ x\ \texttt{= case}\ n^0\ \texttt{of \{ 1 ->}\ chk_{t_j,1}\ \texttt{;}\ \ldots\ \texttt{;}\ n\ \texttt{->}\ chk_{t_j,n}\ \texttt{;}$$
$$n+1\ \texttt{->}\ \texttt{(}n+1\texttt{,}\bar{y}^0\texttt{,}\backslash\bar{z}\texttt{->}x\texttt{)}\ \texttt{\}}$$

We define

$$chk_{t_j,i} = \begin{cases} \texttt{case}\ x^0\ \texttt{of \{}\ p_i\text{–}chk_{t_j}\ \ r_{t_j}\text{–}decl\ \texttt{\}} & \text{if } t_{p_i} \text{ is subtype of } t_j \\ \texttt{(}i\texttt{,(}\backslash\bar{z}\texttt{->}x^0\texttt{),}\bar{y}^0\texttt{)} & \text{otherwise} \end{cases}.$$

If t_j is the type of p_i, then p_i–chk_{t_j} is defined as

$p_i\ \texttt{|}\ g_i\ \texttt{->}\ \texttt{(}i + 1\texttt{,}y^0_{1,1}\texttt{,}\ldots\texttt{,}y^0_{i-1,k_{i-1}}\texttt{,}x_{i,1}\texttt{,}\ldots\texttt{,}x_{i,k_i}\texttt{,}y^0_{i+1,1}\texttt{,}\ldots\texttt{,}y^0_{n,k_n}\texttt{,}\backslash\bar{z}\ \texttt{->}\ z_i\texttt{);}$

Otherwise, p_i–chk_{t_j} is empty. Each r_{t_j}–$decl$ has the form (w_i new variables):

$K_{t_j,1}\ w_1\ \ldots\ w_{a_{j,1}}\ \texttt{->}\ chkrek_{t_j,1}\ \texttt{;}\ \ldots\ \texttt{;}\ K_{t_j,n_j}\ w_1\ \ldots\ w_{a_{j,n_j}}\ \texttt{->}\ chkrek_{t_j,n_j}$

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 $chkrek_{t_j,k} = \texttt{(}n^0\texttt{,(}\backslash\bar{z}\texttt{->}x^0\texttt{),}\bar{y}^0\texttt{)}$. If $a > 0$ we define:

$$chkrek_{t_j,k} = \texttt{let \{ (}n^1\texttt{,}\bar{y}^1\texttt{,}c^1\texttt{) =}\ chk_{t_{l_1}}\ \texttt{(}n^0\texttt{,}\bar{y}^0\texttt{)}\ w_1\ \texttt{\}}$$
$$\texttt{in \{}\ \ldots\ \texttt{let \{ (}n^a\texttt{,}\bar{y}^a\texttt{,}c^a\texttt{) =}\ chk_{t_{l_a}}\ \texttt{(}n^{a-1}\texttt{,}\bar{y}^{a-1}\texttt{)}\ w_a\ \texttt{\}}$$
$$\texttt{in \{ (}n^a\texttt{,}\bar{y}^a\texttt{,}\backslash\bar{z}\ \texttt{->}\ K\ \texttt{(}c^1\ \bar{z}\texttt{)}\ \ldots\ \texttt{(}c^a\ \bar{z}\texttt{)) \}}\ \ldots\ \texttt{\}}$$

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. 6. Semantics of Context–Patterns

into `Haskell`, which also gives us the possibility for an integration of context patterns in the Glasgow Haskell Compiler. Therefore, we added the translation of context patterns to the desugarar phase of the `ghc`. The more efficient approach to implement context patterns by a direct translation to abstract machine code is not adequate for the `ghc`, due to the structure of the intermediate language `Core`.

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].

Appendix: Complete Translation of `initlast`

For the `initlast` example

```
initlast []      = error "Empty list"
initlast (c [x]) = ((c []), x)
```

we obtain the following translation

```
initlast [] = error "Empty list"
initlast l =
  let
    chk0 (n0,xb0) x0 =
      case n0 of
      1 -> case x0 of
           [x] -> (n0+1,x,\z->z)
           []  -> (n0,xb0,\z->x0)
           (w1:w2) ->
                  let (n1,xb1,c1)=chk1 (n0,xb0) w1
                      (n2,xb2,c2)=chk0 (n1,xb1) w2
                  in (n2,xb2,\z->((c1 z):(c2 z)))
      2 -> (2,xb0,\z->z)
    chk1 (n0,xb0) x0 =
      case n0 of
      1 -> (n0,xb0,\z->x0)
      2 -> (n0,xb0,\z->x0)
  in case (chk0 (1,undefined) l) of
     (2,x,c) -> (c [],x)
     _       -> undefined
```

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, *Proceedings 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.

[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.

[HPW92] P. Hudak, S. L. Peyton Jones, and P. Wadler *et. al.* Report on the Programming Language Haskell — A Non-strict, Purely Functional Language. *ACM SIGPLAN Notices*, 27(4), 1992.

[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.

[Moh97] M. Mohnen. Context Patterns in Haskell. In 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, pages 41–58. Springer–Verlag, 1997.

[OW97] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings 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, *Proceedings of the 6th European Symposium on Programming (ESOP)*, number 1058 in LNCS, pages 18–44. Springer–Verlag, 1996.

[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.

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