Pattern Guards and Transformational Patterns

Martin Erwig
Oregon State University
erwig@cs.orst.edu

Simon Peyton Jones
Microsoft Research Ltd, Cambridge
simonpj@microsoft.com

Abstract

We propose three extensions to patterns and pattern matching in Haskell. The first, *pattern guards*, allows the guards of a guarded equation to match patterns and bind variables, as well as to test boolean condition. For this we introduce a natural generalisation of guard *expressions* to guard *qualifiers*.

A frequently-occurring special case is that a function should be applied to a matched value, and the result of this is to be matched against another pattern. For this we introduce a syntactic abbreviation, *transformational patterns*, that is particularly useful when dealing with views.

These proposals can be implemented with very modest syntactic and implementation cost. They are upward compatible with Haskell; all existing programs will continue to work.

We also offer a third, much more speculative proposal, which provides the transformational-pattern construct with additional power to explicitly catch pattern match failure.

We demonstrate the usefulness of the proposed extension by several examples, in particular, we compare our proposal with views, and we also discuss the use of the new patterns in combination with equational reasoning.

1 Introduction

Pattern matching is a well-appreciated feature of languages like ML or Haskell; it greatly simplifies the task of inspecting values of structured data types and facilitates succinct function definitions that are easy to understand. In its basic form, pattern matching tries to identify a certain structure of a value to be processed by a function. This structure is specified by a pattern, and if it can be recovered in a value, corresponding parts of the value are usually bound to variables. These bindings are exploited on the right-hand side of the definition. There are numerous proposals for extending the capabilities of patterns and pattern matching; in particular, the problems with pattern matching on abstract data types have stimulated a lot of research [19, 16, 3, 12, 4, 6, 11]. Other aspects have also been subject to extensions and generalisations of pattern matching [8, 1, 9, 7, 17].

All these approaches differ in what they can be used for, in their syntax, and in their properties, which makes it almost impossible to use two or more different approaches at the same time. Moreover, among all these different approaches there is no clear winner, although so-called *views* seem to be the most prominent and favourite extension.

Therefore, a consolidation of pattern matching at a more fundamental level deserves attention. An extension should be simple enough so that its use is not prohibited by a complex syntax, and it should be powerful enough to express most of the existing approaches.

In this paper we present a proposal for an elementary extension of patterns and pattern matching that naturally extends Haskell's current pattern matching capabilities. The design is influenced by the following goals:

- *Conservative Extension.* Programs that do not use the new feature should not need to be changed and should have unchanged semantics.
- *Simplicity.* We shall not introduce (yet another) more or less complex sub-language for specifying new kinds of patterns, for introducing pattern definitions, and so on. Instead, a minor extension to the syntax with a simple semantics should be aimed at.
- *Expressiveness.* It should be possible to express pattern matching on abstract data types. In particular, views [19, 3, 4, 11], and two kinds of active patterns [12, 6] should be covered.
- *Efficient and Simple Implementation.* The use of the new patterns should not be penalised by longer running times. Moreover, only minimal changes to an existing language should be needed. This facilitates the easy integration of the new concept into existing language implementations and supports a broad evaluation of the concept.

The remainder of this paper is structured as follows: we motivate the need for more powerful pattern matching in Section 2 and present our proposal in Sections 3 and 4. Syntax and semantics are defined in Section 5, and the implementation is discussed in Section 6. A detailed comparison with views is performed in Section 7. In Section 8 we then discuss the use of the new patterns with equational reasoning. A further extension of the expressiveness of our proposal is described in Section 9. Related work is discussed in Section 10, and finally, conclusions are given in Section 11.

2 The need for more powerful pattern matching

In the current version of Haskell pattern matching is not just a straight, one-step process because guards can be used to constrain further the selection of function equations. However, no (additional) bindings can be produced in this second step. This is a somewhat non-orthogonal design, and the extension we propose essentially generalises this aspect.

Consider the following Haskell function definition.

```
filter p []                 = []
filter p (y:ys) | p y       = y : filter p ys
                | otherwise = filter p ys
```

The decision of which right-hand side to choose is made in two stages: first, pattern matching selects a guarded group, and second, the boolean-valued guards select among the right-hand sides of the group.

In these two stages, *only the pattern-matching stage can bind variables, but only the guards can call functions.* It is well known that this design gives rise to a direct conflict between pattern-matching and abstraction, as we now discuss.

2.1 Abstract data types

Consider an abstract data type of sequences, which offers O(1) access to both ends of the sequence (see, for example, [10]):

```
nil   :: Seq a
lcons :: a -> Seq a -> Seq a
lview :: Seq a -> Maybe (a,Seq a)
rcons :: Seq a -> a -> Seq a
rview :: Seq a -> Maybe (Seq a,a)
```

Since sequences are realized as an abstract date type, their representation is not known, and this prohibits the use of pattern matching. The functions `lview` and `rview` provide two views of the sequence, one as a left-oriented list and the other as a right-oriented list, and thus reveal to some degree a representation of sequences (that can be different, though, from their actual implementation). This means that pattern matching against this representation is now principally possible. However, it generally leads to less clearer definitions.

For example, a function to filter such a sequence would have to use a `case` expression to scrutinise the result of, say, `lview`:

```
filtSeq :: (a->Bool) -> Seq a -> Seq a
filtSeq p xs
  = case (lview xs) of
      Nothing -> nil
      Just (y,ys) | p y -> lcons y (filtSeq p ys)
                  | otherwise -> filtSeq p ys
```

This is much less satisfactory than the list version of `filter`, which used pattern-matching directly. Actually, it is possible to write `filtSeq` in a more equational way:

```
filtSeq :: (a->Bool) -> Seq a -> Seq a
filtSeq p xs
  | isJust lv && p y  = lcons y (filtSeq p ys)
  | isJust lv         = filtSeq p ys
  | otherwise         = nil
  where
    lv = lview xs
    Just (y,ys) = lv
```

The auxiliary function `isJust` is taken from the standard library `Maybe`:

```
isJust :: Maybe a -> Bool
isJust (Just x) = True
isJust Nothing  = False
```

The idea here is that the guard `isJust lv` checks that the `lview` returns a `Just` value, while the (lazily-matched) pattern `Just (y,ys)` is only matched if `y` or `ys` is demanded. So now `filtSeq` is more "equational", but it is hardly clearer than before. A well-known approach to reconcile pattern matching and abstract data types is the *views* proposal; we will consider views in detail in Section 7.

2.2 Matching that involves several arguments

As another example, suppose we have an abstract data type of finite maps, with a lookup operation:

```
lookup :: FiniteMap -> Int -> Maybe Int
```

The lookup returns `Nothing` if the supplied key is not in the domain of the mapping, and `(Just v)` otherwise, where `v` is the value that the key maps to. Now consider the following definition:

```
clunky env var1 var2 | ok1 && ok2 = val1 + val2
                     | otherwise  = var1 + var2
where
  m1  = lookup env var1
  m2  = lookup env var2
  ok1 = isJust m1
  ok2 = isJust m2
  Just val1 = m1
  Just val2 = m2
```

Much as with `filtSeq`, the guard `ok1 && ok2` checks that both lookups succeed, using `isJust` to convert the maybe types to booleans. The (lazily matched) `Just` patterns extract the values from the results of the lookups, and bind the returned values to `val1` and `val2`, respectively. If either lookup fails, then `clunky` takes the `otherwise` case and returns the sum of its arguments.

This is certainly legal Haskell, but it is a tremendously verbose and un-obvious way to achieve the desired effect. Is it any better using `case` expressions?

```
clunky env var1 var1 =
  case lookup env var1 of
       Nothing -> fail
       Just val1 ->
         case lookup env var2 of
              Nothing -> fail
              Just val2 -> val1 + val2
where
  fail = var1 + var2
```

This is a bit shorter, but hardly better. Worse, if this was just one equation of `clunky`, with others that follow, then the thing would not work at all. That is, suppose we have

```
clunky' env (var1:var2:vars) | ok1 && ok2
                             = val1 + val2
            where
              m1 = lookup env var1
              ... as before
clunky' env [var1]           = ... some stuff
clunky' env []               = ... more stuff
```

Now, if either of the lookups fail, we want to fall through to the second and third equations for `clunky'`. If we write the definition in the form of a `case` expression, we are forced to make the latter two equations for `clunky'` into a separate definition and call it in the right-hand side of `fail`. This is precisely why Haskell provides guards at all, rather than relying on `if-then-else` expressions: if the guard fails, we fall through to the next equation, whereas we cannot do that with a conditional.

What is frustrating about this is that the solution is so tantalisingly near at hand! What we want to do is to pattern-match on the result of the lookup. We can do it like this:

```
clunky' env vars@(var1:var2:_)
 = help (lookup env var1) (lookup env var2) vars
where
  help (Just v1) (Just v2) vars   = v1 + v2
  help _         _         [var1] = ... some stuff
  help _         _         []     = ... more stuff
```

Now we do get three equations, one for each right-hand side, but it is still clunky. In a big set of equations it becomes hard to remember what each `Just` pattern corresponds to. Worse, we cannot use one lookup in the next. For example, suppose our function was like this:

```
clunky'' env var1 var2 | ok1 && ok2 = val2
                       | otherwise  = var1 + var2
where
  m1   = lookup env var1
  m2   = lookup env (var2 + val1)
  ok1  = isJust m1
  ok2  = isJust m2
  Just val1 = m1
  Just val2 = m2
```

Notice that the second lookup uses `val1`, the result of the first lookup. To express this with a `help` function requires a second helper function nested inside the first. Dire stuff.

2.3 Summary

In this section we shown that Haskell's pattern-matching capabilities are unsatisfactory in certain situations. The first example relates to the well-known tension between pattern-matching and abstraction. The `clunky` example, however, was a little different — there, the matching involved two arguments (`env` and `var1`), and did not arise directly from data abstraction.

There is no fundamental issue of expressiveness: we can rewrite any set of pattern-matching, guarded equations as `case` expressions. Indeed, that is precisely what the compiler does when compiling equations! So should we worry at all? Yes, we should. The reason that Haskell provides guarded equations is because they allow us to *write down the cases we want to consider, one at a time, mostly independently of each other* — the "equational style". This structure is hidden in the `case` version. In the case of `clunky`, two of the right-hand sides are really the same (`fail`).

Furthermore, nested `case` expressions scale badly: the whole expression tends to become more and more indented. In contrast, the equational (albeit verbose) definition, using `isJust` have the merit that they scale nicely to accommodate multiple equations. So we seek a way to accommodate the equational style despite a degree of abstraction.

3 A proposal: pattern guards

Our initial proposal is simple:

> *Instead of being a boolean expression, a guard is a list of qualifiers, exactly as in a list comprehension.*

That is, the only syntax change is to replace *exp* by *quals* in the syntax of guarded equations.

Here is how we would write `clunky`:

```
clunky env var1 var1
   | Just val1 <- lookup env var1
   , Just val2 <- lookup env var2
   = val1 + val2
   ... other equations for clunky
```

The semantics should be clear enough. The qualifiers are matched in order. For a `<-` qualifier, which we call a *pattern guard*, the right-hand side is evaluated and matched against the pattern on the left. If the match fails, then the whole guard fails, and the next equation is tried. If it succeeds, then the appropriate binding(s) are made, and the next qualifier is matched, in the augmented environment. Unlike list comprehensions, however, the type of the expression to the right of the `<-` is the same as the type of the pattern to its left. The bindings introduced by pattern guards scope over all the remaining guard qualifiers, and over the right-hand side of the equation. If there is a `where` clause, then its bindings scope over all the guards, just as in Haskell at present.

Just as with list comprehensions, boolean expressions can be freely mixed with the pattern guards. For example:

```
f x | [y] <- x
    , y > 3
    , Just z <- h y
    = ...
```

Haskell's current guards therefore emerge as a special case, in which the qualifier list has just one element, a boolean expression.

Just as with list comprehensions, a `let` qualifier can introduce a binding. It is also possible to do this with pattern guards with a simple variable pattern `a <-` *exp*. However a `let` qualifier is a little more powerful, because it can introduce a recursive or mutually-recursive binding. It is not clear whether this power is particularly useful, but it seems more uniform to have exactly the same syntax as list comprehensions.

One could argue that the notation `<-` is misleading, suggesting the idea of *drawn from* as in a list comprehension. But it is very nice to reuse precisely the list-comprehension syntax. Furthermore, the only viable alternative is `=`, and that would lead to parsing difficulties, because we rely on the `=` to herald the arrival of the right-hand side of the equation. Consider `f x | y = h x = 3`.

Using pattern guards for `filtSeq` exposes a small deficiency:

```
filtSeq :: (a->Bool) -> Seq a -> Seq a
filtSeq p xs
  | Just (y,ys) <- lview xs, p y = lcons y (filtSeq p ys)
  | Just (y,ys) <- lview xs      = filtSeq p ys
  | otherwise                    = nil
```

There is the annoying repeated call to `lview xs` (which can be shared by putting it in a `where` clause), plus the annoying repeated pattern match (which cannot). Maybe one would like some kind of nested guards, thus:

```
filtSeq :: (a->Bool) -> Seq a -> Seq a
filtSeq p xs
  | Just (y,ys) <- lview xs
      | p y       = lcons y (filtSeq p ys)
      | otherwise = filtSeq p ys
  | otherwise     = nil
```

Such an extension would make perfect sense, and would not be hard to implement, but its power-to-weight ratio is significantly lower than for our main proposal: it is less often useful (less power), and requires more new syntax (greater weight).

It is also possible to write `filtSeq` using an auxiliary function `filtSeq'`:

```
filtSeq :: (a->Bool) -> Seq a -> Seq a
filtSeq p = filtSeq' p . lview

filtSeq' :: (a->Bool) -> Maybe (a,Seq a) -> Seq a
filtSeq' p Nothing       = nil
filtSeq' p (Just (y,ys)) =
        if p y then lcons y (filtSeq p ys)
               else filtSeq p ys
```

However, this introduces an indirection into the definition and the "solution" that is offered by this approach to the problem of pattern matching with abstract data types is simply to avoid it.

4 A further proposal: transformational patterns

Pattern guards allow the programmer to call an arbitrary function and pattern-match on the result. In the important special case addressed by views, these calls take a very stylised form, and this motivates us to propose some special syntax, *transformational patterns*, in support.

Here is how we might write `filtSeq`, using a transformational pattern:

```
filtSeq :: (a->Bool) -> Seq a -> Seq a
filtSeq p (Just (y,ys))!lview
            | p y       = lcons y (filtSeq p ys)
            | otherwise = filtSeq p ys
filtSeq p Nothing!lview = nil
```

The transformational pattern `(Just (y,ys))!lview` means informally "apply `lview` and match against `Just (y,ys)`". The expression to the right of the "`!`" is called *pattern action*. Transformational patterns are simply syntactic sugar for an equivalent form using pattern guards, but they are notationally a little more concise. Furthermore, they are quite like views: "match `Just (y,ys)` against the `lview` view of the argument".

Since the function in a transformational pattern can refer to any variables that are in scope in, or bound by, the `where` clause, we can write `clunky` as:

```
clunky env (Just val1)!(lookup env)
           (Just val2)!(lookup env) = val1 + val2
           ... other equations for clunky
```

This gives transformational patterns just a little more power than views, at the cost of a somewhat *ad hoc* flavour. To summarise, transformational patterns help to keep function equations single-lined, which greatly enhances readability and understanding of function definitions containing several equations. Moreover, transformational patterns are particularly useful when simulating views, see Section 7, and with equational reasoning, see Section 8.

5 Syntax and semantics

Based on the Haskell 98 Report [14], we need two small changes to the syntax to integrate pattern guards and transformational patterns: first, a guard is not just anymore given by an expression but by a list of qualifiers, and an atomic pattern can be a pattern extended by an expression:

gd	$\rightarrow$	$\mid\ qual_1,\ \ldots,\ qual_n$	Pattern Guard
$apat$	$\rightarrow$	$\ldots$	
	$\mid$	$apat!\,aexp$	Transformational Pattern

We define the semantics of pattern guards and transformational patterns by a series of equations that relate them to "ordinary" `case` expressions of Haskell.

We start with the reduction of pattern guards to nested `case` expressions. For this, we first unfold multiple guards in matches to nested `case` expressions. This is done to keep the further translation manageable because guards themselves can be lists of qualifiers. Hence, we replace rule (c) by

```
(c') case v of { p | g1 -> e1 ...
                   | gn -> en where { decls };
                 _      -> e' }
     = case e' of { y ->
         case v of { p ->
            let decls in
              case () of {
                () | g1 -> e1;
                _       -> ... case () of {
                                 () | gn -> en;
                                 _       -> y }...}
            _ -> y }}
         where y is a completely new variable
```

The construction `case () of () -> ...` indicates that these `case` expressions do not pattern matching, but are just used to look at the guards.

Next we expand a list of qualifiers of each guard into a nested `case` expression.

```
(s) case () of { () | q1, ..., qn -> e; _ -> e' }
     = case e' of { y ->
         case () of {
           () | q1 -> ... case () of {
                            () | qn -> e;
                            _       -> y };...
           _       -> y }}
         where y is a completely new variable
```

The next three equations explain how qualifiers are resolved: (t) boolean guards are transformed into conditionals, (u) a local declaration can be just moved into the body, and (v) generators are again transformed into `case` expressions where the generating expression is scrutinised and matched against the binding pattern.

```
(t) case () of { () | e0 -> e; _ -> e' }
     = if e0 then e else e'

(u) case () of { () | let decls -> e; _ -> e' }
     = let decls in e

(v) case () of { () | (p0 <- e0) -> e; _ -> e' }
     = case e0 of { p0 -> e; _ -> e' }
```

It remains to reduce transformational patterns to pattern guards. This is done by the following equation:

```
(w) case v of { p!f -> e; _ -> e' }
     = case v of { x | (p <- f x) -> e; _ -> e' }
       where x is a completely new variable
```

6 Implementation

The standard technology used by compilers for generating efficient matching trees from sets of equations can be adapted straightforwardly to accommodate pattern guards and transformational patterns. Currently, pattern guards are fully implemented in GHC, and transformational patterns are not yet implemented.

The efficiency issue is a little more pressing than with pure pattern matching, because the access functions, called in the transformational pattern or the pattern guard, may be arbitrarily expensive. For example, consider

```
data AbsInt = Pos Int | Neg Int

absInt :: Int -> AbsInt
absInt n = if n>=0 then Pos n else Neg (-n)

f :: Int -> Int
f (Pos n)!absInt = n+1
f (Neg n)!absInt = -(n+1)
```

This is reasonably concise. But how many times is `absInt` called? In this case, it is pretty clear that it need only be called once. But what about this:

```
g ((Pos a)!absInt : as) []                      = ...
g []                    ((Pos b)!absInt : bs)   = ...
g ((Neg a)!absInt : as) ((Neg b)!absInt : bs)   = ...
g _                     _                       = ...
```

Now it gets harder to tell! In general, it may be necessary to know the pattern-match compilation algorithm used by the compiler in order to reason precisely about how many times `absInt` will be called. Nevertheless, it is not difficult to expand a pattern matching algorithm by knowledge about transformational patterns so that in cases like above (when only a function and not a complex expression is used as a pattern action) one can ensure a translation into nested `case` expressions so that at each argument position each pattern action is invoked only once.

Without any changes to the pattern matching algorithm, pattern guards allow us to express the sharing explicitly in some cases:

```
f n | Pos n' <- abs_n = n'+1
    | Neg n' <- abs_n = -(n'+1)
    where
      abs_n = absInt n
```

Here the common definition `abs_n` makes clear that there should only be one call to `absInt`. But a `where` clause cannot always be used. For example, consider the following silly function `g` using the `AbsInt` view (for an explanation of views, see the next section)

```
view AbsInt of Int = Pos Int | Neg Int
     where absInt = ... as above ...

g (Pos (Pos n)) = n+1
g (Pos (Neg n)) = -(n+1) -- A bit silly
```

Now we have to write

```
g n | Pos n1 <- abs_n, Pos n2 <- absInt n1 = n2+1
    | Pos n1 <- abs_n, Neg n2 <- absInt n1 = -(n2+1)
    where
      abs_n = absInt n
```

We can share the first call to `absInt` but not the second because `n1` is not in the scope of the `where` clause. Instead we have to rely on (a) knowledge of the pattern-compilation algorithm, and (b) common-subexpression elimination, to justify our hopes that there will be only two calls to `absInt`.

Again, in transformational-pattern form, the shared computations seem easier to spot and can be ensured by an extension of the pattern matching algorithm:

```
g (Pos (Pos n)!absInt)!absInt = n+1
g (Pos (Neg n)!absInt)!absInt = -(n+1)
```

7 A comparison with views

As we have already discussed, pattern guards and transformational patterns can be seen as addressing a similar goal to that of views, namely reconciling pattern matching with data abstraction. Views were proposed by Wadler ages ago [19], and are the subject of a concrete proposal for a Haskell language extension [4].

Consider, for example, the cartesian representation of complex numbers with a polar view.

```
data Complex = Cart Float Float

view Polar of Complex = Polar Float Float
     where
       polar (Cart r i) =
             Polar (sqrt(r*r+i*i)) (atan2 r i)

pole = Polar
```

With this view, function definitions can be written using the polar representation in patterns relying on the automatic application of the view transformation `polar`. This means that it is possible to assume a particular representation of abstract data types in function definitions without exposing or even knowing the representation used in the implementation. Note that in addition to the view transformation `polar` we also need a function `pole` to construct values of the view type.

Without views we define the view type as an ordinary data type and the view transformation as a function.

```
data Polar = Polar Float Float

polar (Cart r i) = Polar (sqrt(r*r+i*i)) (atan2 r i)
```

Now it is natural to ask whether views subsume pattern guards or vice versa. The answer is "neither".

7.1 Do views subsume pattern guards?

The views proposal [4] points out that you can use views to simulate (some) guards and, as we saw above, views have similar purpose and functionality to at least some applications of pattern guards.

However, views give a view on a *single* value, whereas guards allow arbitrary function calls to combine in-scope values. For example, `clunky` matches `(Just val1)` against `(lookup env var1)`. We do not want a view of `env` nor of `var1` but rather of their combination by `lookup`. Views simply do not help with `clunky`.

Views are capable of dealing with the data abstraction issue of course. However, each conditional selector would require its own view, complete with its own view type. This can seem rather heavyweight. For example, our Haskell compiler, GHC, has an abstract data type called `Type`. The `Type` module offers the function

```
getFunTyMaybe :: Type -> Maybe (Type,Type)
```

This function is a mixture of a predicate ("is this a function type") and a view ("the argument and result type are these"). To use views we would have to say:

```
view FunType of Type = FunType Type Type
                     | NotFunType
  where
    funType (Fun arg res) = FunType arg res
    funType other_type    = NotFunType
```

This seems a bit heavyweight (three new names instead of one) compared with re-using the existing `Maybe` type. Not only does this reuse save defining new types, but it allows the existing library of functions on `Maybe` types to be applied directly to the result of `getFunTyMaybe`.

Sometimes it is quite unclear whether a function should be regarded as a "view" or not. For example, GHC has another function on `Type`:

```
tyvarsOf :: Type -> [TyVar]
```

that returns the free type variables of a type. One could regard this as specifying a "view" of a type as the free variables of the type:

```
view TyVarsOf of Type = TyVarsOf [TyVar]
  where
    tyVarsOf ty = ...
```

Now we could write

```
f :: Type -> Int
f (TyVarsOf tyvars) = length tyvars
```

instead of

```
f :: Type -> Int
f ty = length (tyvarsOf ty)
```

But the "view" has not made the program any simpler; indeed one could claim the reverse.

Our point is this: there is a continuum between functions that one might regard as "views" of a type and other functions that are predicates, or selectors, or property-extractors. Extending the language with views undesirably forces the programmer to choose whether a particular function is best regarded as a view or not. Pattern guards do not force such a choice. Instead, the same access function (for example, `getFunTyMaybe`) can be used in ordinary expressions or pattern guards.

7.2 Do pattern guards subsume views?

Views allow the treatment of an abstract data type as a concrete data type as far as pattern matching is concerned. In particular, the same notation can be used for pattern matching on abstract and on concrete data types. In contrast, with transformational patterns we have to extend the patterns by pattern actions.

Although views are notationally only a little more concise than transformational patterns, one could argue that the small notational difference presents a usage barrier, and therefore views might be better suited to promote data abstraction while retaining pattern matching.

Now with regard to the `Polar` example, with views we can simply use a `Polar` constructor in a pattern knowing that the view transformation is applied automatically, whereas with transformational patterns we have to call the `polar` function explicitly.

Thus we might prefer to write (using views)

```
multC (Polar r1 t1) (Polar r2 t2)
      = pole (r1*r2) (t1+t2)
```

rather than (using transformational patterns)

```
multC (Polar r1 i1)!polar (Polar r2 i2)!polar
      = Polar (r1*r2) (t1+t2)
```

One might argue, though, that the latter accurately indicates that there may be some work involved in matching against a view, compared to ordinary pattern matching. With transformational patterns we can also safely use the `Polar` constructor (see also Section 8).

7.3 Summary

We believe that the pattern-guard and transformational-pattern proposal

- is much simpler to specify and implement than views;
- gets some expressiveness that is simply inaccessible to views;
- successfully reconciles pattern matching with data abstraction, albeit with a slightly less compact notation than views;
- is less heavyweight to use when defining many information extraction functions over an ADT;
- does not conceal where computation takes place.

8 Equational reasoning with transformational patterns

Problems in reasoning with Miranda laws and views arise because a constructor of a lawful type or view can be used both within a pattern and as an expression. In particular, the use in an expression is problematic. Suppose, for example, we have a constructor `Half :: Int -> Half` of a view type `Half` that is defined to divide its argument by 2.

```
view Half of Int = Half Int
     where half i = Half (i 'div' 2)
```

When we now define a function

```
f :: Half -> Int
f (Half i) = i+1
```

we could try to reason, by replacing equals for equals, that `f (Half 8)` = `9`, which, however, is not true because `f (Half 8)` = `5` due to the computational part of `Half`.

The solution proposed by Burton and Cameron [3] is to forbid the use of view constructors, such as `Half`, in expressions. This works well, but one always has to be aware of the status of a constructor and whether it is a view constructor or not.

With transformational patterns we would have to define a plain data type together with a function performing the desired computation of the constructor `Half`.

```
data Half = Half Int

half i = Half (i 'div' 2)
```

Now when we use `Half` in equations, nothing harmful can happen because all computation is made explicit. For example, it is valid to conclude `f (Half 8)` = `9` since `Half` is just a data type constructor performing no computation on its argument at all. This is because the above definition for `f` essentially does not use the view in its original sense. To make use of the view computations we have to give a different definition for `f`:

```
f :: Half -> Int
f (Half i)!half = i+1
```

But now it is evident that we just cannot use an equation like `f (Half 8)!half` = `9` because `(Half 8)!half` is not an expression. Hence, the additional requirement made by Burton and Cameron is implicitly given in our approach just by the syntax of transformational patterns.

On the other hand, it *is* possible to use transformational patterns in equational reasoning. To explain this it is helpful to recall how patterns are used, for example, in the transformation of an expression `f e`. This happens in two steps: first, the structure of `e` is examined (this is either obvious because `e` is an application of a constructor or it is given in a precondition of the current transformation, for example, something like `e` = `x:xs`). Then the equation for `f` that matches the structure of `e` is determined, say `f (x:xs) = e'`, and `f e` is substituted by `e'`.

Now transformational patterns fit into this scheme as follows. Suppose, `f` contains an equation `f p!c = e'`. Then when you can make the assumption `p` = `c e` in a proof, you can replace the expression `f e` by `e'`. Of course, as with other patterns, one has to choose the first possible match to get a sound transformation.

We illustrate this by a small example. Suppose we have defined selection sort with the help of a function `min' :: Ord a => [a] -> (a,[a])` that extracts a minimum from a list.

```
sort :: Ord a => [a] -> [a]
sort []          = []
sort (m,r)!min' = m:sort r
```

Now we would like to prove the correctness of `sort`. Using the prelude function `all` we can define a predicate for sorted lists as follows.

```
sorted []     = True
sorted (x:xs) = all (x<=) xs && sorted xs
```

Suppose we already know the following property of `min'`.

Lemma 1 `min' l` = `(x,xs)` ⇒ `all (x<=) xs` = `True`

Then we can easily prove:

Lemma 2 `sorted (sort l)` = `True`

Proof. We perform a case analysis on the list argument: if `l` = `[]`, we have

```
  sorted (sort [])
= sorted []
= True
```

If `l` ≠ `[]`, we have `min' l` = `(m,r)`, and we can select the second equation for `sort`:

```
  sorted (sort l)
= sorted (m:sort r)
= all (m<=) r && sorted r
```

Now by induction `sorted r` can assumed to be true, and `all (m<=) r` is true due to Lemma 1. □

9 Pattern match failure as a first-class entity?

In Haskell, pattern match failure is considered a local event that has to be dealt with in one and the same `case` expression or (if not dealt with) results in an (unrecoverable) error. An alternative, more general view is that a failing pattern match raises an exception, say, `Fail`, that can be caught in the same *or in another* `case` expression. Such a treatment of pattern match failure could be nicely exploited by transformational patterns: a pattern matching failure in a pattern action can be caught to be able to reach the next function equation. In this section we sketch a possible extension along these lines. We regard this section as much more speculative than the rest of the paper.

As a first example we give a definition for the view-version of the function `power` [19].

```
power :: Int -> Int -> Int
power x 0        = 1
power x n!asOdd  = x*power (x*x) n
power x n!asEven =   power (x*x) n
```

The two functions `asOdd` and `asEven` both halve their argument but match only for odd and even numbers:

```
asOdd  n | n `mod` 2 == 1 = n `div` 2
asEven n | n `mod` 2 == 0 = n `div` 2
```

With these partial definitions, the application of, say, `asOdd` to an even number, is defined to yield a program error that propagates through to the top level, but with the extended `Fail` semantics the pattern match failure raised in `asOdd` is caught in `power` and causes the third equation to be tried.

As a further example we show how transformational patterns make it possible to use pattern matching successfully on graphs. Graph algorithms traditionally consider graphs as monolithic blocks, and this view is reflected in their clumsy definitions. As an alternative we have proposed an inductive view of graphs that makes recursive definitions of graph algorithms feasible [5]: a graph is either empty, or it is constructed by adding a node together with edges from/to its predecessors/successors. Let `Node` be the type of node values, and let `Graph` be the type of unlabelled graphs. A *node context* is a node together with a list of successors (third tuple component) and a list of predecessors (first component):

```
type Context = ([Node],Node,[Node])
```

Then we can define an empty graph constant and a function for successively adding node contexts:

```
empty :: Graph
embed :: (Context,Graph) -> Graph
```

Note that `embed` yields a runtime error if either the node to be inserted is already present in the graph or if any of the predecessor or successor nodes does not exist in the graph. We also have a function `match` that locates the context of a particular node in a graph.[1]

```
match :: Node -> Graph -> (Context,Graph)
```

[1] `match` is defined so that for all nodes `v` contained in `g` the following law holds: `embed (match v g)` = `g`.

Now with a function suc that simply projects onto the third component of a context we can give a highly concise definition of depth-first search:

```
dfs :: [Node] -> Graph -> [Node]
dfs []     _                = []
dfs (v:vs) (c,g)!(match v) = v:dfs (suc c++vs) g
dfs (_:vs) g                = dfs vs g
```

The arguments of dfs are a list of nodes to be visited and the graph to be searched, and the result is a list of nodes in depth-first order. Note that we do not need a data structure for remembering the nodes that we have already seen — by repeatedly removing contexts from the graph we rather "forget" (in the graph) the nodes that have been visited so far. If we then try to revisit a node, this leads in match to a match failure, causing dfs to try the third equation, which simply ignores the current node.

9.1 Semantics

Note that catching pattern matching failure is *not* possible with transformational patterns when they are just reduced to pattern guards. Therefore, we have to provide an independent semantics definition. One possible way to go is to define pattern matching within a Haskell version that accounts for exceptions. A proposal for exceptions was made in [13], giving a precise semantics together with an efficient implementation. Pattern match failure could be defined in that context to raise a Fail exception, and pattern matching had to catch Fail exceptions to select function equations. In that proposal catching exceptions leads, in general, to non-determinism. To see this, consider the expression e1 + e2, and assume that e1 and e2 result in two different exceptions. Now what should be the result of e1 + e2? If we avoid to fix the evaluation order, all we can do is to either define + to make a non-deterministic choice or to return the set of all exceptions raised anywhere within e1 and e2. The last proposal was made in [13]. Even when dealing with exception sets, checking for a particular exception re-introduces non-determinism. However, in a framework where Fail is the only exception one could also think of just checking whether an exception has occurred at all or not. This eliminates non-determinism to a large degree, but it might still be the case that one and the same program could diverge or not depending on, for example, the platform or the larger context in which it was compiled. Again, consider expressions like bot + Fail.

Another possibility is to define the more general behaviour of transformational patterns within the current Haskell framework. The problem we face here is that a pattern-match failure within the pattern action must not yield $\perp$ since this cannot be caught in the case expression containing the transformational pattern. We can cope with this by performing a source-code transformation $e^{\mathcal{M}}$ of pattern actions to wrap all possible results with Just, and add a default case that returns Nothing. Then we perform pattern matching against Nothing in the case rule dealing with transformational patterns. The corresponding case equation is easy to give:

(w') $\texttt{case}\ v\ \texttt{of \{}\ p\texttt{!}f\ \texttt{->}\ e\texttt{;}\ \texttt{\_ ->}\ e'\ \texttt{\}}$
$\quad = \texttt{case}\ e'\ \texttt{of \{}\ y\ \texttt{->}$
$\qquad \texttt{case}\ f^{\mathcal{M}}\ v\ \texttt{of \{}$
$\qquad\quad \texttt{Nothing ->}\ y\texttt{;}$
$\qquad\quad \texttt{Just}\ x\ \texttt{-> case}\ x\ \texttt{of \{}\ p\ \texttt{->}\ e\texttt{;}\ \texttt{\_ ->}\ y\ \texttt{\}\}\}}$
where x and y are completely new variables

It remains to be shown how pattern actions can be lifted into the Maybe type. We give a definition that follows the structure of expressions. Since only case expressions are a source of possible pattern match failure, wrapping Just and adding Nothing happens just there. In all other cases, lifting is just recursively passed through (or ignored). In particular, constants, constructors, and variables remain unchanged. For application, abstraction, and case expressions we obtain:

$$\begin{array}{lcl}(e_1\ e_2)^{\mathcal{M}} & = & e_1^{\mathcal{M}}\ e_2 \\ (\lambda x.e)^{\mathcal{M}} & = & \lambda x.e^{\mathcal{M}} \\ (\texttt{case}\ e\ \texttt{of}\ \{p_i\,\texttt{->}\,e_i\})^{\mathcal{M}} & = & \texttt{case}\ e\ \texttt{of \{}\ \{p_i\ \texttt{-> Just}\ e_i\}\texttt{;} \\ & & \qquad\qquad\ \texttt{\_ \ \ -> Nothing \}}\end{array}$$

The problem with this approach is that it does not work well with separate compilation, in particular, with precompiled libraries: in general, we do not have access to the definition of a function that is used in a pattern action and that lives in a separately compiled module. In that case the above scheme breaks down because we do not know the function's source code and we therefore cannot apply the source code transformation.

9.2 Summary

In this section we have sketched a more speculative development of the pattern-matching idea. We regard it as debatable whether the additional power of these extended transformational patterns is worth the cost in terms of semantic complications, or loss of separate compilation. However, first-class pattern-matching failure would very much obviate the need for pattern guards because Fail can be used to "step back" into a function's pattern matching process.

Our earlier proposals, of pattern guards and transformational patterns, described in Sections 3 and 4, do not involve any such semantic or compilation complications.

10 Related work

One of the first extensions to pattern matching was the lawful types of Miranda [18, 16]: in this approach the programmer is allowed to add equations to a data type definition that act as rewrite rules to transform data type values into a canonical representation. The approach has two main problems: first, in many applications different representations are needed to use data types and pattern matching conveniently (see, for example, the polar vs. cartesian representation of complex values [3, 12] or differently rooted trees to represent sets [6]), and Miranda laws prevent this. Second, laws cause problems with equational reasoning [16].

The most prominent and most widely accepted extension to pattern matching seems to be the *view* mechanism which was first proposed by Phil Wadler [19] and that was later adapted by several others [3, 4, 11]. With views one can have as many different representations of a data type as needed. For each such representation, called *view*, two functions `in` and `out` must be defined that map from the (main) data type into the view type and vice versa. Views do not suffer from the first restriction of Miranda laws, but the view transformations must be inverses of each other, and this sometimes either leads to partial definitions or causes problems with equational reasoning due to ambiguities.

The reasoning problems were first solved by Burton and Cameron [3] who restrict the use of view constructors only to patterns. Hence, the `out` function is not needed anymore, and the `in` function need not be injective. This has been adopted by all view proposals that were made since then.

Okasaki [11] has defined the view concept (the proposal of [4]) for Standard ML. He pays special attention to the interaction of view transformations with stateful computations that are possible in ML.

We have demonstrated that views can be easily simulated by transformational patterns.

In [12] *active destructors* were introduced that allow the definition and use of patterns, called *active patterns*, that might perform computations to produce bindings: `c p match q where r = e` defines an active destructor `c p` that can be used as a pattern in place of `q`. During the matching process `e` is evaluated using bindings produced by `q` and producing new bindings in `r` that can finally be used by `p`. Active destructors extend the capabilities of views, but they require even more syntactic overhead. In particular, a new notation is needed for the typing of patterns. Active destructors can, to some degree, perform computations like pattern guards and transformational patterns, but they cannot access other variables bound in the same function equation, which we consider a highly useful feature.[2]

Active destructors can also be simulated by pattern guards and transformational patterns. For example, define a function `c q = e` and use the transformational pattern `p!c` in place of the active destructor `c p`.

The "`p as f`" construction introduced in [2] is also similar to transformational patterns: a pattern matching function `f` can be converted into a pattern `p` so that it can be composed with other patterns. This is used to express pattern matching on union types.

The goal of the *active patterns* introduced in [6][3] was to enable the matching of specific representations of data type values. Whereas views always map a value in one view type to a canonical representation, active patterns allow the selection of an arbitrary one. The idea is that specialised constructors can reorganise data type values before they are matched. This reorganisation is intended to yield a representation that suits the current function definition best. With regard to these preprocessing capabilities, active patterns are similar to transformational patterns and also to active destructors. However, active patterns are more general than active destructors because their computing functions have access to other bindings of the pattern, and active patterns are less general than active destructors and than pattern guards and transformational patterns because the argument and result type must be the same.

Just as views and laws and the other approaches mentioned so far were motivated by combining pattern matching and ADTs, there are some other, more limited, approaches that are also driven by specific applications: *context patterns* [9] give direct access to arbitrary deeply nested sub-parts of terms; they are very similar to other tree transforming languages (for example, [8, 15]). In particular, they only work for algebraic, free data types, and computations on matched values are not possible. The *abstract value constructors* presented in [1] provide a facility to abbreviate terms and allow the use of these abbreviations as expressions as well as within patterns. Again, no computations are possible on the matched values. In contrast, *pattern abstractions* [7] do allow a very limited form of computation; the aim is to generalise pattern matching only as far as static analyses, such as checking overlapping patterns or exhaustiveness, are still decidable.

A different route to pattern matching is taken by Tullsen [17], who considers patterns as functions of type `a -> Maybe b`. This allows the treatment of patterns as first-class objects; in particular, it is possible to write pattern combinators. Although the semantics of patterns can be simplified considerably by that approach, the *use* of patterns in the language is rather clumsy even after the introduction of some syntactic sugar through so-called *pattern binders*.

Let us finally compare the described extensions with our proposal from a general point of view. Whereas it is quite easy and straightforward to use a pattern guard or a transformational pattern in a function definition (just put it there!), the use of an active destructor (or of a view, or of any other proposal) requires the definition of such a pattern at some other place before it can be used. In many cases this is prohibitive either because adding an additional declaration is not justified by, say, only one application or because it is just faster or shorter not to use that concept. For example, the definition of the function `last` based on a reverse view of lists (see [19]) requires some effort to define the view type and the view transformations, whereas it can be immediately written using pattern guards:

```
last xs | (x:_) <- reverse xs = x
```

or with transformational patterns:

```
last (x:_)!reverse = x
```

The situation here is comparable to that of function definitions vs. anonymous functions (that is, lambda-abstractions): sometimes it is useful to have anonymous functions, although it is syntactically nicer to apply a defined function.

A further aspect is that pattern definitions can be used only for pattern matching, whereas a function used in a pattern guard or as a pattern action can also be used in expressions.

Another difference we already stressed is that our proposal makes it explicit where computations take place in patterns

[2] Active destructors allow a very limited and rather ad-hoc way of passing additional parameters into pattern functions; this is described in [12] only for Haskell-specific arithmetic $n+k$-patterns and requires yet another extension to the typing notation.

[3] The work of [12] and [6] was performed independently leading to the homonym.

whereas this information is hidden by views or active destructors. As far as programming with ADTs is concerned, hiding might be in most cases appropriate, however, in some cases it might be more convenient to have explicit information about computations instead of defining, learning and remembering many different views.

11 Conclusions

We have introduced pattern guards and transformational patterns that offer the possibility of preprocessing values before they are matched against patterns. Our proposal covers and even generalises previous approaches to pattern matching on abstract data types. This extension allows a useful class of programs to be written much more elegantly than with the current version of Haskell.

The required extensions to existing languages are minimal. This applies both to the syntax and to the implementation, which makes the introduction into Haskell very easy. In fact, pattern guards are already fully implemented in GHC (transformational patterns are not yet implemented). The syntax is not as neat as for views, but rather makes the computations that happen explicit. We have argued that this can be an advantage in using the patterns, understanding programs, and in equational reasoning.

Our proposal is attractive also from a more general language design point of view because in formulating recursive function definitions one always has to make two design decisions: (i) which arguments are needed in which form for the recursive function application(s) on the right-hand side, and (ii) how can the parameters from the left-hand side be brought into the required form. With traditional patterns, these two closely related design decisions are separated by moving (ii) into a (possibly distant) `where` block. With transformational patterns both activities have been brought closer together; this facilitates the programmer to focus the view on the essential parts of recursive definitions.

Fähndrich and Boyland [7] call their pattern matching extension *pattern abstractions*. Their approach (and all the others) require the naming of pattern abstractions. Since this is not needed in our approach, pattern guards and transformational patterns can be viewed as complementing the landscape of pattern matching extensions by *anonymous pattern abstractions*.

Acknowledgements

We thank Chris Okasaki and the anonymous reviewers for their helpful comments.

References

[1] W. E. Aitken and J. H. Reppy. Abstract Value Constructors. In *ACM Workshop on ML and its Applications*, pages 1–11, 1992.

[2] P. Buneman and B. Pierce. Union Types for Semistructured Data. Technical Report MS-CIS-99-09, University of Pennsylvania, 1999.

[3] F. W. Burton and R. D. Cameron. Pattern Matching with Abstract Data Types. *Journal of Functional Programming*, 3(2):171–190, 1993.

[4] F. W. Burton, E. Meijer, P. Sansom, S. Thompson, and P. Wadler. Views: An Extension to Haskell Pattern Matching. http://haskell.org/development/views.html, 1996.

[5] M. Erwig. Inductive Graphs and Functional Graph Algorithms. *Journal of Functional Programming*. To appear.

[6] M. Erwig. Active Patterns. In *8th Int. Workshop on Implementation of Functional Languages*, LNCS 1268, pages 21–40, 1996.

[7] M. Fähndrich and J. Boyland. Statically Checkable Pattern Abstractions. In *2nd ACM Int. Conf. on Functional Programming*, pages 75–84, 1997.

[8] R. Heckmann. A Functional Language for the Specification of Complex Tree Transformations. In *European Symp. on Programming*, pages 175–190, 1988.

[9] M. Mohnen. Context Patterns in Haskell. In *8th Int. Workshop on Implementation of Functional Languages*, LNCS 1268, pages 307–319, 1996.

[10] C. Okasaki. Simple and Efficient Purely Functional Queues and Deques. *Journal of Functional Programming*, 5(4):583–592, 1995.

[11] C. Okasaki. Views for Standard ML. In *ACM Workshop on ML and its Applications*, pages 14–23, 1998.

[12] P. Palao Gostanza, R. Peña, and M. Núñez. A New Look at Pattern Matching in Abstract Data Types. In *1st ACM Int. Conf. on Functional Programming*, pages 110–121, 1996.

[13] S. L. Peyton Jones, A. Reid, T. Hoare, S. Marlow, and F. Henderson. A Semantics for Imprecise Exceptions. In *ACM Conf. on Programming Languages Design and Implementation*, pages 25–36, 1999.

[14] S. L. Peyton Jones, J. Hughes *et al.* Report on the Programming Language Haskell 98. 1999.

[15] C. Queinnec. Compilation of Non-Linear, Second Order Patterns on S-Expressions. In *2nd Int. Symp. on Programming Language Implementation and Logic Programming*, LNCS 456, pages 340–357, 1990.

[16] S. Thompson. Lawful Functions and Program Verification in Miranda. *Science of Computer Programming*, 13:181–218, 1990.

[17] M. Tullsen. First Class Patterns. In *2nd Int. Workshop on Practical Aspects of Declarative Languages*, LNCS 1753, pages 1–15, 2000.

[18] D. A. Turner. Miranda: A Non-strict Functional Language with Polymorphic Types. In *Conf. on Functional Programming and Computer Architecture*, LNCS 201, pages 1–16, 1985.

[19] P. Wadler. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. In *14th ACM Symp. on Principles of Programming Languages*, pages 307–313, 1987.