View Patterns in GHC

Dan Licata Simon Peyton-Jones

MSR Cambridge

Pattern Matching and Abstract Types

It is common to define an abstract type:

```
type Seq a
empty :: Seq a
<|    :: a → Seq a → Seq a
|>    :: Seq a → a → Seq a
...
```

Pattern Matching and Abstract Types

And a concrete view of it for pattern matching:

```
type Seq a
...
data ViewL a = EmptyL
             | a :< (Seq a)
viewl :: Seq a → ViewL a
```

Pattern Matching and Abstract Types

And a concrete view of it for pattern matching:

```
type Seq a
...
data ViewL a = EmptyL
             | a :< (Seq a)
viewl :: Seq a → ViewL a
```

But using the view is a little inconvenient

Using the View

Case instead of equations:

```
map f s = case viewl s of
     EmptyL → empty
     x :< xs → f x <| map f xs
```

Using the View

Case instead of equations:

```
map f s = case viewl s of
     EmptyL → empty
     x :< xs → f x <| map f xs
```

Or use pattern guards:

```
map f s | EmptyL  <- viewl s = empty
map f s | x :< xs <- viewl s = f x <| map f xs
```

Using the View

Case instead of equations:

```
map f s = case viewl s of
     EmptyL → empty
     x :< xs → f x <| map f xs
```

Or use pattern guards:

```
map f s | EmptyL  <- viewl s = empty
map f s | x :< xs <- viewl s = f x <| map f xs
```

But neither of these nest well

View Patterns to the Rescue

Idea: apply a function inside a pattern:

```
map f (viewl → EmptyL)  = empty
map f (viewl → x :< xs) = f x <| map f xs
```

View Patterns to the Rescue

Idea: apply a function inside a pattern:

```
map f (viewl → EmptyL)  = empty
map f (viewl → x :< xs) = f x <| map f xs
```

```
prs :: Seq a → Seq (a,a)
prs(v → EmptyL)                      = empty
prs(v → x :< (v → EmptyL))          = empty
prs(v → x :< (v → x’ :< xs)) = (x,x’) <| prs xs
v = viewl
```

View Patterns to the Rescue

Or even, using an extension we'll talk about later:

```
prs :: Seq a → Seq (a,a)
prs(→ EmptyL)                        = empty
prs(→ x :< (→ EmptyL))             = empty
prs(→ x :< (→ x' :< xs)) = (x,x') <| prs xs
```

View Patterns in GHC

1. What are view patterns?

2. How do you use them?

3. How are they implemented?

View Patterns in GHC

1. **What are view patterns?**

2. How do you use them?

3. How are they implemented?

View Patterns

New form of pattern: `(expr → pat)`

Typing: If `expr` has type `A → B`
and `pat` matches a `B`
then `(expr → pat)` matches an `A`.

View Patterns

New form of pattern: `(expr → pat)`

Typing: If `expr` has type `A → B` and `pat` matches a `B` then `(expr → pat)` matches an `A`.

Evaluation: To match `(expr → pat)` against `v`, match `pat` against `(expr v)`.

View Patterns

New form of pattern: `(expr → pat)`

Typing: If `expr` has type `A → B` and `pat` matches a `B` then `(expr → pat)` matches an `A`.

Evaluation: To match `(expr → pat)` against `v`, match `pat` against `(expr v)`.

Scoping: The variables bound by `(expr → pat)` are the variables bound by `pat`.

View Patterns

New form of pattern: `(expr → pat)`

Typing: If `expr` has type `A → B` and `pat` matches a `B` then `(expr → pat)` matches an `A`.

Evaluation: To match `(expr → pat)` against `v`, match `pat` against `(expr v)`.

Scoping: The variables bound by `(expr → pat)` are the variables bound by `pat`.

But what's in scope in `expr`?

Scoping

It's useful for "earlier" variables to be bound "later" in the pattern.

Parametrized views:

```
bits :: Int → ByteStr → Maybe (Word, ByteStr)

parsePacket :: Int → ByteStr → ...
parsePacket n (bits n → Just (hdr, bs)) = ...
```

Scoping

It's useful for "earlier" variables to be bound "later" in the pattern.

Pattern synonyms/first-class patterns:

```
f :: (A → Maybe B) → A → ...
f g (g → Just n) = ...
```

Scoping

Rule: variables to the left (in tuples, constructors, curried arguments) are in scope

OK	BAD
`(x, x → y)`	`(x → y, x)`
`c x (x → y)`	`c (x → y) x`
`f x (x -> y) = ...`	`f (x → y) x = ...`

Scoping

But expressions in `let` bindings may not refer to other bindings from the same `let`.

OK

```
let x = ... in
  let (x -> y) = ... in y
```

BAD

```
let x = ... in
    (x -> y) = ... in y
```

(More on this later)

One Little Extension

Writing the view expression can be tiresome:

```
prs :: Seq a → Seq (a,a)
prs(v → EmptyL)                  = empty
prs(v → x :< (v → EmptyL))       = empty
prs(v → x :< (v → x' :< xs)) = (x,x') <| prs xs
```

```
v = viewl
```

One Little Extension

Writing the view expression can be tiresome:

```
prs :: Seq a → Seq (a,a)
prs(v → EmptyL)                  = empty
prs(v → x :< (v → EmptyL))       = empty
prs(v → x :< (v → x' :< xs)) = (x,x') <| prs xs
v = viewl
```

Can we avoid writing it some of the time?

Implicit View Function

Define a type class

```
class View a b where
    view :: a → b
```

Then `(→ pat)` means `(view → pat)`

Implicit View Function

Define a type class

```
class View a b where
   view :: a → b
```

Then `(→ pat)` means `(view → pat)`

```
instance View (Seq a) (ViewL a) where
  view = viewl
...
prs(→ x :< (→ x' :< xs)) = (x,x') <| prs xs
```

Implicit View Function

Define a type class

```
class View a b where
   view :: a → b
```

- Add instances for the "canonical" views of abstract types
- Maybe a functional dependency in one direction or the other? Otherwise infer

```
prs :: ∀a,b. View a (ViewL b) =>
           a -> Seq (b,b)
```

And That’s It

One new form of pattern, and one new type class in the prelude

- No new form of declaration (e.g. ’view’ or ’pattern synonym’)
- View expressions are ordinary Haskell functions: don’t need to be written with view patterns in mind (e.g., Data.Sequence) and can be called from ordinary Haskell code
- No changes to import or export mechanisms
- Static and dynamic semantics are simple

View Patterns in GHC

1. What are view patterns?

2. **How do you use them?**

3. How are they implemented?

Join lists

```
data JList a = Empty
             | Single a
             | Join (JList a) (JList a)
data JListView a = Nil | Cons a (JList a)
```

The view is used in its own definition:

```
...
view (Join (view -> Cons xh xt) y) =
   Cons xh (Join xt y)
view (Join (view -> Nil) y) = view y
```

Partial Views

Use `Maybe`-targeted views for pattern-matching ad-hoc data such as XML or strings:

```
ifs :: String -> Maybe Integer
ffs :: String -> Maybe Float
```

```
add (ifs -> Just n, ifs -> Just n’) = ...
add (ffs -> Just f, ffs -> Just f’) = ...
add _    = print “whoops, bad string”
```

Other (ab)uses

Both patterns:

```
both :: a -> (a,a)
both x = (x,x)
f (both -> (xs, h : t)) = h : (xs ++ t)
```

Iterator style:

```
map f [] = []
map f (x : (map f -> xs)) = f x : xs
```

Other (ab)uses

Both patterns:

```
both :: a -> (a,a)
both x = (x,x)
f (both -> (xs, h : t)) = h : (xs ++ t)
```

Iterator style:

```
map f [] = []
map f (x : (map f -> xs)) = f x : xs
```

See the GHC Wiki for more idioms (n+k patterns, named constants,...)

View Patterns in GHC

1. What are view patterns?

2. How do you use them?

3. **How are they implemented?**

Static Semantics

GHC checks lexical scoping in a pass called the renamer, before type checking

- Patterns were not already in the recursive loop with expressions

- Some plumbing needed to change to deliver the appropriate contexts for checking view expressions

Type checking was comparatively easy!

Desugaring into Core

GHC compiles pattern matching using the matrix algorithm in the SPJ/Wadler chapter of [SPJ'87].

1. Match a matrix of patterns

$$\begin{array}{c|c} p_{11} & \ldots \\ \hline \vdots & \\ \hline p_{1n} & \ldots \end{array}$$

   against a vector of variables $(x_1, \ldots)$

2. Identify the maximal group of rows from the top whose leftmost patterns can be put into the same case statement.

Desugaring into Core

View patterns with the same expression can be put in the same case. When top maximal group is

$e \longrightarrow p_1$	$\ldots$
$\vdots$	
$e \longrightarrow p_n$	$\ldots$

1. Recursively match $(x', \ldots)$ against

p_1	$\ldots$
$\vdots$	
p_n	$\ldots$

2. Wrap (`let` x' `=` $e\ x$ `in` $\ldots$) around it

Efficiency of Generated Code

So view functions that line up in a column only get applied once:

```
prs :: Seq a → Seq (a,a)
prs(v → EmptyL)                    = empty
prs(v → x :< (v → EmptyL))    = empty
prs(v → x :< (v → x' :< xs)) = (x,x') <| prs xs
```

desugars into the 2 applications of v that you'd write explicitly

View Patterns in GHC

1. What are view patterns?

2. How do you use them?

3. How are they implemented?

Related Work

View patterns have been implemented in HaMLet-S [Rossberg], Humlock [Murphy et al.], and F# [Syme et al.]

Lots of other proposals for views/pattern synonyms:

Wadler	Burton et al.	Okasaki	Erwig
Palao et al.	Odersky et al.	Reppy et al.	Tullsen

. . .

See the GHC Wiki for discussion and comparison

View patterns

1. Make it a little easier to pattern-match abstract types
2. Provide a sort of first-class pattern as well
3. Are a simple extension that's easy to implement

Will be in GHC HEAD within the next couple of weeks

Thanks for listening!