Through a Glass, Abstractly

Lenses and the Power of Abstraction

Adam Gundry

Haskell eXchange
October 13, 2017

Well-Typed
The Haskell Consultants

The plan

- Optics as an abstract interface
- A concrete implementation: the `lens` library
- Introducing abstraction: the `optics` library

What is a lens?

Formation:

$$\mathrm{Lens}\ (s :: \star)\ (a :: \star) :: \star$$

Introduction:

$$\mathit{lens} :: (s \to a) \to (s \to a \to s) \to \mathrm{Lens}\ s\ a$$

Elimination:

$$\begin{aligned}\mathit{view} &:: \mathrm{Lens}\ s\ a \to s \to a\\ \mathit{set} &:: \mathrm{Lens}\ s\ a \to a \to s \to s\end{aligned}$$

Computation:

$$\begin{aligned}\mathit{view}\ (\mathit{lens}\ f\ g)\ \ \ \ \ s &= f\ s\\ \mathit{set}\ \ \ (\mathit{lens}\ f\ g)\ a\ s &= g\ s\ a\end{aligned}$$

Example of a lens

$$\mathit{firstly} :: \mathsf{Lens}\ (a, x)\ a$$
$$\mathit{firstly} = \mathit{lens}\ \mathit{fst}\ (\lambda(\_, y)\ x \to (x, y))$$

$$\mathit{view}\ \mathit{firstly}\ \ \ \ (\texttt{'a'}, \texttt{'b'}) = \texttt{'a'}$$
$$\mathit{set}\ \mathit{firstly}\ \texttt{'c'}\ (\texttt{'a'}, \texttt{'b'}) = (\texttt{'c'}, \texttt{'b'})$$

Why do we care?

- Lenses are first-class values

```
useLens :: Lens s Int → ...

data FooLens s = MkFooLens (Lens s Int)

class HasFoo s where
  foo :: Lens s Int
```

Why do we care?

- Lenses are first-class values

$$\mathit{useLens} :: \mathrm{Lens}\ s\ \mathrm{Int} \to \ldots$$

$$\mathbf{data}\ \mathrm{FooLens}\ s = \mathrm{MkFooLens}\ (\mathrm{Lens}\ s\ \mathrm{Int})$$

$$\mathbf{class}\ \mathrm{HasFoo}\ s\ \mathbf{where}$$
$$\quad \mathit{foo} :: \mathrm{Lens}\ s\ \mathrm{Int}$$

- Lenses compose:

$$(\circ) :: \mathrm{Lens}\ r\ s \to \mathrm{Lens}\ s\ a \to \mathrm{Lens}\ r\ a$$

$$\mathit{firstly} \circ \mathit{firstly} :: \mathrm{Lens}\ ((a, x), y)\ a$$

There's more to life than lenses

Formation:

$$\text{Getter}\ (s :: \star)\ (a :: \star) :: \star$$

Introduction:

$$to :: (s \to a) \to \text{Getter}\ s\ a$$

Elimination:

$$view :: \text{Getter}\ s\ a \to s \to a$$

Computation:

$$view\ (to\ f)\ s = f\ s$$

There's more to life than lenses

Formation:

$$\text{Setter}\ (s :: \star)\ (a :: \star) :: \star$$

Introduction:

$$\mathit{sets} :: ((a \to a) \to s \to s) \to \text{Setter}\ s\ a$$

Elimination:

$$\mathit{over} :: \text{Setter}\ s\ a \to (a \to a) \to s \to s$$

Computation:

$$\mathit{over}\ (\mathit{sets}\ f)\ g\ s = f\ g\ s$$

Well-Typed

Subtyping

- Every Lens gives us a Setter
- Every Lens gives us a Getter
- A Getter or Setter doesn't give us a Lens
- It's rather like Lens is a subtype of Getter and Setter
- But Haskell doesn't have subtyping... does it?

$$\begin{aligned}
&\mathit{firstly} :: \mathrm{Lens}\ (a, x)\ a \\
&\mathit{over} \quad :: \mathrm{Setter}\ s\ a \to (a \to a) \to s \to s
\end{aligned}$$

$$\mathit{over}\ \mathit{firstly} :: (a \to a) \to (a, x) \to (a, x)$$

Composition

How can we give a type to composition, such that:

- Composing optics gives us the most general possible result
- Composing incompatible optics gives us a nice error?

$\mathit{firstly} \circ \mathit{firstly} :: \mathsf{Lens}\ ((a, x), y)\ a$

$\mathit{firstly} \circ \mathit{to\ not} :: \mathsf{Getter}\ (\mathsf{Bool}, x)\ \mathsf{Bool}$

$\mathit{sets\ map} \circ \mathit{to\ not}$ -- type error

The `lens` library in one complicated slide

$$
\begin{aligned}
&\textbf{type}\ \text{Lens}\ \ s\ a = \textbf{forall}\ f\ .\ \text{Functor}\ f \Rightarrow \\
&\qquad\qquad (a \to f\ a) \to s \to f\ s \\
&\textbf{type}\ \text{Getter}\ s\ a = \textbf{forall}\ f\ .\ (\text{Contravariant}\ f, \text{Functor}\ f) \Rightarrow \\
&\qquad\qquad (a \to f\ a) \to s \to f\ s \\
&\textbf{type}\ \text{Setter}\ s\ a = \textbf{forall}\ f\ .\ \text{Settable}\ f \Rightarrow \\
&\qquad\qquad (a \to f\ a) \to s \to f\ s
\end{aligned}
$$

$(\circ) :: (b \to c) \to (a \to b) \to a \to c$

$(\circ) = (\,.\,)$ -- yes, that's (.) from the Prelude

What's right with `lens`?

- It works
- Rich source of new optic flavour discoveries
- Automatic "subtyping"
- Extremely composable: more polymorphism than you can shake a stick at
- Can write optics without depending on the `lens` package

What's wrong with `lens`?

- Too transparent:

$$\mathit{firstly} :: \mathrm{Functor}\ f \Rightarrow (a \to f\ a) \to (a, x) \to f\ (a, x)$$

$$\begin{aligned}\mathit{firstly}\ .\ \mathit{to}\ \mathit{not} :: {} & (\mathrm{Contravariant}\ f, \mathrm{Functor}\ f) \Rightarrow \\ & (\mathrm{Bool} \to f\ \mathrm{Bool}) \to (\mathrm{Bool}, x) \to f\ (\mathrm{Bool}, x)\end{aligned}$$

What's wrong with `lens`?

- Too transparent:

$$\mathit{firstly} :: \mathrm{Functor}\ f \Rightarrow (a \to f\ a) \to (a, x) \to f\ (a, x)$$

$$\mathit{firstly} \mathbin{.} \mathit{to}\ \mathit{not} :: (\mathrm{Contravariant}\ f, \mathrm{Functor}\ f) \Rightarrow (\mathrm{Bool} \to f\ \mathrm{Bool}) \to (\mathrm{Bool}, x) \to f\ (\mathrm{Bool}, x)$$

- Too expressive:

$$\mathit{sets}\ \mathit{map} \mathbin{.} \mathit{to}\ \mathit{not} :: (\mathrm{Contravariant}\ f, \mathrm{Settable}\ f) \Rightarrow (\mathrm{Bool} \to f\ \mathrm{Bool}) \to [\mathrm{Bool}] \to f\ [\mathrm{Bool}]$$

-- Nothing can be both Contravariant and Settable

Well-Typed

Notorious lens errors

set (*to fst*)

Notorious lens errors

set (*to fst*)

```
* No instance for (Contravariant Identity)
    arising from a use of ‘to’
* In the first argument of ‘set’, namely ‘(to fst)’
```

The dreaded monomorphism restriction

let *f* = *firstly* **in**
let *p* = (`'a'`, `'b'`) **in**
(*view f p*, *set f* `'c'` *p*)

The dreaded monomorphism restriction

let *f* = *firstly* **in**
let *p* = (`'a'`, `'b'`) **in**
(*view f p*, *set f* `'c'` *p*)

```
* Couldn't match type 'Const Char' with 'Identity'
  Expected type:
      ASetter (Char, Char) (Char, Char) Char Char
    Actual type:
      (Char -> Const Char Char)
      -> (Char, Char) -> Const Char (Char, Char)
* In the first argument of 'set', namely 'f'
```

Design goals for the `optics` library

Can we imagine a library that:

- uses opaque abstractions for optics
- gives good type inference properties
- retains subtyping
- allows us to express (only) ”good” compositions of optics?

Defining a subtype hierarchy

```
data A_Lens
data A_Getter
data A_Setter
```

```
class k `Is` l where ...
instance k `Is` k where ...
instance A_Lens `Is` A_Getter where ...
instance A_Lens `Is` A_Setter where ...
```

Defining optics

$$\textbf{newtype}\ \mathrm{Optic}\ k\ s\ a = \mathrm{Optic}\ \{\ldots\}$$

$$\begin{aligned}
&\textbf{type}\ \mathrm{Lens} &&= \mathrm{Optic}\ \mathrm{A_Lens} \\
&\textbf{type}\ \mathrm{Getter} &&= \mathrm{Optic}\ \mathrm{A_Getter} \\
&\textbf{type}\ \mathrm{Setter} &&= \mathrm{Optic}\ \mathrm{A_Setter}
\end{aligned}$$

$$sub :: k\ \text{`Is`}\ l \Rightarrow \mathrm{Optic}\ k\ s\ a \rightarrow \mathrm{Optic}\ l\ s\ a$$

Subtyping in practice

$$\mathit{firstly} :: \mathrm{Lens}\ (a, x)\ a$$
$$\mathit{over} \ \ :: \mathrm{Setter}\ s\ a \rightarrow (a \rightarrow a) \rightarrow s \rightarrow s$$
$$\mathit{over}\ (\mathit{sub}\ \mathit{firstly}) :: (a \rightarrow a) \rightarrow (a, x) \rightarrow (a, x)$$

We (probably) don't want to write *sub* at every call site!

$$\mathit{over} :: k\ \text{`Is`}\ \mathrm{A\_Setter} \Rightarrow \mathrm{Optic}\ k\ s\ a \rightarrow (a \rightarrow a) \rightarrow s \rightarrow s$$

Composition of optics

```
class (k `Is` m, l `Is` m) => Join k l m | k l -> m
```

```
instance Join m m m
instance Join A_Lens A_Getter A_Getter
  -- lots of similar instances omitted
```

```
(∘) :: Join k l m => Optic k r s -> Optic l s a -> Optic m r a
```

Well-Typed

Type inference

With `lens`:

$$\mathit{firstly} :: \mathrm{Functor}\ f \Rightarrow (a \to f\ a) \to (a, x) \to f\ (a, x)$$

$$\begin{aligned}\mathit{firstly}\ .\ \mathit{to}\ \mathit{not} :: {} & (\mathrm{Contravariant}\ f, \mathrm{Functor}\ f) \Rightarrow \\ & (\mathrm{Bool} \to f\ \mathrm{Bool}) \to (\mathrm{Bool}, x) \to f\ (\mathrm{Bool}, x)\end{aligned}$$

With `optics`:

$$\begin{aligned}\mathit{firstly} & :: \mathrm{Optic}\ \mathrm{A\_Lens}\ (a, x)\ a \\ \mathit{firstly} \circ \mathit{to}\ \mathit{not} & :: \mathrm{Optic}\ \mathrm{A\_Getter}\ (\mathrm{Bool}, x)\ \mathrm{Bool}\end{aligned}$$

Well-Typed

Catching invalid compositions

sets map $\circ$ *to not*

```
* No instance for (Join A_Setter A_Getter m)
    arising from a use of ‘%’
* In the expression: sets map % to not
```

Catching invalid compositions

sets map $\circ$ *to not*

```
* No instance for (CanCompose A_Setter A_Getter)
    arising from a use of ‘%’
* In the expression: sets map % to not
```

Notorious lens errors redux

set (*to fst*)

Notorious lens errors redux

set (*to fst*)

With `lens`:

```
* No instance for (Contravariant Identity)
    arising from a use of ‘to’
* In the first argument of ‘set’, namely ‘(to fst)’
```

Notorious lens errors redux

set (*to fst*)

With `lens`:

```
* No instance for (Contravariant Identity)
    arising from a use of ‘to’
* In the first argument of ‘set’, namely ‘(to fst)’
```

With `optics`:

```
* No instance for (Is A_Getter A_Setter)
    arising from a use of ‘set’
* In the expression: set (to fst)
```

Well-Typed

Dodging the dreaded monomorphism restriction

This just works, because our Lens type isn't polymorphic:

$\textbf{let}\ f = \mathit{firstly}\ \textbf{in}$
$\textbf{let}\ p = (\texttt{'a'}, \texttt{'b'})\ \textbf{in}$
$(\mathit{view}\ f\ p, \mathit{set}\ f\ \texttt{'c'}\ p)$

Advantages of this approach

- Types that say what they mean
- More comprehensible type errors
- Less vulnerable to the monomorphism restriction
- Free choice of lens implementation

The inevitable drawbacks

- Polymorphism has to be ”baked in” rather than emerging naturally from definitions
- (We haven’t implemented indexed lenses, yet, for example)
- Can’t insert points into the subtyping order post hoc
- Number of instances required is quadratic in number of optic flavours

The core library trade-off

- Can write optics without depending on `lens`
- This is not the case for `optics`
- Instead, we offer `optics-core` with minimal extra dependencies
- If using a library that defines `lens`-style optics, it's easy to convert them

Conclusions

- Polymorphism is a trade-off, not a holy grail
- Abstraction helps build comprehensible interfaces
- Consider how type inference will help or hinder your users

`https://github.com/well-typed/optics`

Thank you

Well-Typed

One possible implementation of the abstraction

```
type family Constraints (k :: ⋆) (f :: ⋆ → ⋆) :: Constraint
type instance Constraints A_Lens   f = (Functor f)
type instance Constraints A_Getter f = (Contravariant f,
                                        Functor f)
type instance Constraints A_Setter f = (f ∼ Identity)
```

```
newtype Optic k s a =
  Optic (forall f . Constraints k f ⇒ (a → f a) → s → f s)
```

Well-Typed

One possible implementation of the abstraction

```
class k `Is` l where
  implies :: (Constraints k f => r) -> (Constraints l f => r)
```

```
instance k `Is` k where
  implies = id

instance A_Lens `Is` A_Getter where
  implies = id
```

One possible implementation of the abstraction

$sub :: \textbf{forall}\ k\ l\ s\ a\,.\ k\ \text{`Is`}\ l \Rightarrow \mathrm{Optic}\ k\ s\ a \rightarrow \mathrm{Optic}\ l\ s\ a$
$sub\ (\mathrm{Optic}\ o) = \mathrm{Optic}\ (implies'\ o)$
$\quad \textbf{where}$
$\qquad implies' :: \textbf{forall}\ f\,.\ \mathrm{Optic\_\_}\ k\ f\ s\ a \rightarrow \mathrm{Optic\_\_}\ l\ f\ s\ a$
$\qquad implies'\ x = implies\ \texttt{@}k\ \texttt{@}l\ \texttt{@}f\ x$

$\textbf{type}\ \mathrm{Optic\_\_}\ k\ f\ s\ a =$
$\qquad \mathrm{Constraints}\ k\ f \Rightarrow (a \rightarrow f\ a) \rightarrow s \rightarrow f\ s$

One possible implementation of the abstraction

$$(\circ) :: \mathrm{Join}\ k\ l\ m \Rightarrow \mathrm{Optic}\ k\ r\ s \to \mathrm{Optic}\ l\ s\ a \to \mathrm{Optic}\ m\ r\ a$$
$$f \circ g = \mathit{sub}\ f\ \text{`}\mathit{compose}\text{`}\ \mathit{sub}\ g$$

$$\mathit{compose} :: \mathrm{Optic}\ k\ r\ s \to \mathrm{Optic}\ k\ s\ a \to \mathrm{Optic}\ k\ r\ a$$
$$\mathrm{Optic}\ o\ \text{`}\mathit{compose}\text{`}\ \mathrm{Optic}\ o' = \mathrm{Optic}\ (o\,.\,o')$$

Enough with the backup slides, already!

In `lens`, *view* is defined for Fold, which is more general than a Getter. In particular, every Traversal is a Fold. Calling *view* on a Fold imposes a Monoid constraint:

view traverse [Just `"abc"`, Nothing, Just `"def"`]
Just `"abcdef"`

toListOf traverse [Just `"abc"`, Nothing, Just `"def"`]
[Just `"abc"`, Nothing, Just `"def"`]