COMP4075/G54RFP: Lecture 16
Optics

Henrik Nilsson

University of Nottingham, UK

Guest Tutorial: Preparations (1)

- Ben Clifford: Build a RESTful Room-Booking Server Using Servant and Aeson
  Fri. 6 Dec 2019, 11:00–13:00, CS A32

Guest Tutorial: Preparations (1)

- Ben Clifford: Build a RESTful Room-Booking Server Using Servant and Aeson
  Fri. 6 Dec 2019, 11:00–13:00, CS A32
- Goal: Building simple booking system accessible through a JSON+HTTP API using established Haskell libraries.

Guest Tutorial: Preparations (1)

- Ben Clifford: Build a RESTful Room-Booking Server Using Servant and Aeson
  Fri. 6 Dec 2019, 11:00–13:00, CS A32
- Goal: Building simple booking system accessible through a JSON+HTTP API using established Haskell libraries.
- Hands on tutorial! Preferably, bring laptop with:
  - Stack (cross-platform Haskell dev. system)
  - Tutorial prerequisites installed
  - WiFi connectivity

Guest Tutorial: Preparations (2)

- See link off guest lecture webpage for details:
  `https://github.com/benclifford/2019-nottingham-prereq`

Guest Tutorial: Preparations (2)

- See link off guest lecture webpage for details: `https://github.com/benclifford/2019-nottingham-prereq`
- To get most out of the tutorial, it is essential to:
  - Bring a laptop with prerequisites installed
  - Resolve issues ***before*** the tutorial

Optics: What?

- ***Optics*** are ***functional references***: focusing on one part of a structure for access an update.

Optics: What?

- ***Optics*** are ***functional references***: focusing on one part of a structure for access an update.
- Examples of "optics" include ***Lens***, ***Prism***. ***Iso***, ***Traversable***.

Optics: What?

- ***Optics*** are ***functional references***: focusing on one part of a structure for access an update.
- Examples of “optics” include ***Lens***, ***Prism***. ***Iso***, ***Traversable***.
- Different kinds of “optics” allow different number of focal points and may or may not be invertible.

Optics: What?

- ***Optics*** are ***functional references***: focusing on one part of a structure for access an update.
- Examples of “optics” include ***Lens***, ***Prism***. ***Iso***, ***Traversable***.
- Different kinds of “optics” allow different number of focal points and may or may not be invertible.
- Today, we’ll look at lenses. Lenses ***compose*** very nicely, allowing focusing on the target step-by-step.

Motivating Example (1)

- Haskell's "records" often get critisized.

Motivating Example (1)

- Haskell's "records" often get critisized.
- Somewhat undeserved:
  - Merit of simplicity
  - Disciplined field naming conventions can mitigate some of the drawbacks

Motivating Example (1)

- Haskell's "records" often get critisized.
- Somewhat undeserved:
  - Merit of simplicity
  - Disciplined field naming conventions can mitigate some of the drawbacks
- Lenses go a long way to address other criticisms.

Motivating Example (2)

$$
\begin{array}{l}
\textbf{data}\ Point = Point\ \{ \\
\quad positionX :: Double, \\
\quad positionY :: Double \\
\} \\
\textbf{data}\ Segment = Segment\ \{ \\
\quad segmentStart :: Point, \\
\quad segmentEnd\ \ :: Point \\
\}
\end{array}
$$

Motivating Example (3)

Field access is straightforward.
For example, given $seg :: Segment$:

$$end\_y = positionY \circ segmentEnd \mathbin{\$} seg$$

Motivating Example (3)

Field access is straightforward.
For example, given $seg :: Segment$:

$$end\_y = positionY \circ segmentEnd \ \$ \ seg$$

Field update is much clunkier:

$$\begin{array}{l}
\mathbf{let}\ end = segmentEnd\ seg \\
\mathbf{in}\ seg\,\{\,segmentEnd = \\
\qquad\qquad end\,\{\,positionY = 2 * positionY\ end\,\} \\
\quad\ \}
\end{array}$$

Lenses to the rescue! (1)

Lenses for focusing on specific fields can be defined manually, but there is support for automating the process which is convenient if there are many fields.

Field names must then start by an underscore.

Lenses to the rescue! (2)

```
import Control.Lens

data Point = Point {
    _positionX :: Double,
    _positionY :: Double
}
makeLenses ''Point

data Segment = Segment {
    _segmentStart :: Point,
    _segmentEnd   :: Point
}
makeLenses ''Segment
```

Lenses to the rescue! (3)

This gives us lenses for the fields:

$$
\begin{aligned}
&positionX &&:: Lens'\ Point\ Double\\
&positionY &&:: Lens'\ Point\ Double\\
&segmentStart &&:: Lens'\ Segment\ Point\\
&segmentEnd &&:: Lens'\ Segment\ Point
\end{aligned}
$$

Lenses to the rescue! (3)

This gives us lenses for the fields:

$$
\begin{aligned}
&\mathit{positionX} &&:: \mathit{Lens'}\ \mathit{Point}\ \mathit{Double}\\
&\mathit{positionY} &&:: \mathit{Lens'}\ \mathit{Point}\ \mathit{Double}\\
&\mathit{segmentStart} &&:: \mathit{Lens'}\ \mathit{Segment}\ \mathit{Point}\\
&\mathit{segmentEnd} &&:: \mathit{Lens'}\ \mathit{Segment}\ \mathit{Point}
\end{aligned}
$$

Individual fields can now be accessed and updated:

$$
\begin{aligned}
&\mathit{view}\ \mathit{segmentEnd}\ \mathit{seg}\\
&\mathit{set}\ \mathit{segmentEnd}\ \mathit{seg}
\end{aligned}
$$

Lenses to the rescue! (4)

But what is really cool is that lenses compose!

Ordinary function composition, but note the order: from “large” to “small”:

$$view\ (segmentEnd \circ positionY)\ seg$$
$$over\ (segmentEnd \circ positionY)\ (2*)\ seg$$

How does this work? (1)

$Lens'\ a\ b$ is a type synonym:

$$\begin{array}{l} \textbf{type}\ Lens'\ s\ a = \\ \quad Functor\ f \Rightarrow (a \to f\ a) \to (s \to f\ s) \end{array}$$

How does this work? (1)

$Lens'\ a\ b$ is a type synonym:

$$\mathbf{type}\ Lens'\ s\ a = \\ Functor\ f \Rightarrow (a \rightarrow f\ a) \rightarrow (s \rightarrow f\ s)$$

This is a function that transforms an operation on a part of type a of a structure of type s to an operation on the whole structure.

How does this work? (2)

In particular:

$$
\begin{array}{l}
\mathit{positionY} :: \\
\quad \mathit{Functor}\ f \Rightarrow \\
\quad\quad (\mathit{Double} \to f\ \mathit{Double}) \to (\mathit{Point} \to f\ \mathit{Point}) \\
\mathit{segmentEnd} :: \\
\quad \mathit{Functor}\ f \Rightarrow \\
\quad\quad (\mathit{Point} \to f\ \mathit{Point}) \to (\mathit{Segment} \to f\ \mathit{Segment})
\end{array}
$$

And thus:

$$
\begin{array}{l}
\mathit{segmentEnd} \circ \mathit{positionY} :: \mathit{Functor}\ f \Rightarrow \\
\quad (\mathit{Double} \to f\ \mathit{Double}) \to (\mathit{Segment} \to f\ \mathit{Segment})
\end{array}
$$

How does this work? (3)

Combinators like $view$, set, $over$ instantiate the functor to something suitable to achieve the desired effect:

$$
\begin{aligned}
&set \quad :: ASetter\ s\ t\ a\ b \rightarrow b \rightarrow s \rightarrow t \\
&over \ :: ASetter\ s\ t\ a\ b \rightarrow (a \rightarrow b) \rightarrow s \rightarrow t \\
&\textbf{type}\ ASetter\ s\ t\ a\ b = \\
&\qquad\qquad (a \rightarrow Identity\ b) \rightarrow s \rightarrow Identity\ t
\end{aligned}
$$

Consequently, e.g.:

$$
\begin{aligned}
&over\ (segmentEnd \circ positionY) :: \\
&\quad (Double \rightarrow Double) \rightarrow Segment \rightarrow Segment
\end{aligned}
$$

How does this work? (4)

$$view :: MonadReader\ s\ m \Rightarrow Getting\ a\ s\ a \to m\ a$$

$$\textbf{type}\ Getting\ r\ s\ a = (a \to Const\ r\ a) \to s \to Const\ r\ s$$

$Const$ is the constant functor:

$$\textbf{newtype}\ Const\ a\ b = Const\ \{\, getConst :: a\,\}$$

Consequently, e.g.:

$$view\ (segmentEnd \circ positionY) :: MonadReader\ Segment\ m \Rightarrow m\ Double$$

How does this work? (5)

As $(\rightarrow)\ Segment$ is a reader monad,

$$\begin{aligned} &view\ (segmentEnd \circ positionY) :: \\ &\quad MonadReader\ Segment\ m \Rightarrow m\ Double \end{aligned}$$

is just a function $Segment \rightarrow Double$.

Some other useful lenses

The *Lens* package defines lots of optics for standard types. In particular, it defines lenses for all field of tuples up to size 19. For example:

$> view\ \_2\ (3,4,5)$

4

$> view\ \_2\ (5,6,7,8)$

6

$> set\ \_3\ 9\ (5,6,7,8)$

$(5,6,9,8)$

$> over\ \_3\ (*2)\ (3,4,5)$

$(3,4,10)$