Optics in the abstract

Adam Gundry

ZuriHac, June 2021 —

Well-Typed
The Haskell Consultants

A brief word from our sponsor

- Well-Typed is a Haskell consultancy company, established in 2008
- Team of about 18 Haskell experts

- On-site and remote training courses
- Haskell software development and consulting
- GHC maintenance, development and support

- `info@well-typed.com`

Well-Typed

The plan

Today:

1. An introduction to `optics`
2. Working with nested data
3. Duplicate Record Fields
4. Reflections on library design

Tomorrow:

- **Understanding memory usage with eventlog2html and ghc-debug** by Matthew Pickering and Ben Gamari

Well-Typed

Join in

#optics channel on Discord

`https://github.com/well-typed/optics-zurihac-2021`

Part I

An introduction to `optics`

The `optics` library

We'll focus on the `optics` family of packages, by Andrzej Rybczak, Andres Löh, Oleg Grenrus and myself.

Optics in Haskell go back much further, with major contributions by Twan van Laarhoven, Russell O'Connor and Edward Kmett.

What is a lens?

Type formation:

$$\mathsf{Lens}\ (s :: \mathsf{Type})\ (t :: \mathsf{Type})\ (a :: \mathsf{Type})\ (b :: \mathsf{Type}) :: \mathsf{Type}$$

Introduction:

$$\mathit{lens} :: (s \to a) \to (s \to b \to t) \to \mathsf{Lens}\ s\ t\ a\ b$$

Elimination:

$$\begin{aligned} \mathit{view}_L &:: \mathsf{Lens'}\ s\ a \to s \to a \\ \mathit{set} &:: \mathsf{Lens}\ s\ t\ a\ b \to b \to s \to t \end{aligned}$$

Computation and laws:

$$\begin{aligned} \mathit{view}_L\ (\mathit{lens}\ f\ g)\ \ \ s &\equiv f\ s \\ \mathit{set}\ (\mathit{lens}\ f\ g)\ b\ s &\equiv g\ s\ b \end{aligned}$$

$$\begin{aligned} \mathit{view}_L\ l\ (\mathit{set}\ l\ b\ s) &\equiv b \\ \mathit{set}\ l\ (\mathit{view}_L\ l\ s)\ s &\equiv s \\ \mathit{set}\ l\ c\ (\mathit{set}\ l\ b\ s) &\equiv \mathit{set}\ l\ c\ s \end{aligned}$$

Example of a lens

$$
\begin{aligned}
&\mathit{frst} :: \mathsf{Lens}\ (a, x)\ (b, x)\ a\ b \\
&\mathit{frst} = \mathit{lens}\ \mathit{fst}\ (\lambda(\_, y)\ x \to (x, y))
\end{aligned}
$$

Example of a lens

$$
\begin{aligned}
&\mathit{frst} :: \mathsf{Lens}\ (a, x)\ (b, x)\ a\ b \\
&\mathit{frst} = \mathit{lens}\ \mathit{fst}\ (\lambda(\_, y)\ x \to (x, y))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{view}_L\ \mathit{frst}\qquad (\texttt{'a'}, \texttt{'b'}) \equiv \texttt{'a'} \\
&\mathit{set}\ \mathit{frst}\ \texttt{'c'}\ (\texttt{'a'}, \texttt{'b'}) \equiv (\texttt{'c'}, \texttt{'b'})
\end{aligned}
$$

Type-changing update

The *set* operation can change the type of the value stored in the structure, and hence the type of the structure itself.

For example:

$$\mathit{set}\ \mathit{frst}\ \mathsf{True}\ (\texttt{'a'}, \texttt{'b'}) \equiv (\mathsf{True}, \texttt{'b'}) :: (\mathsf{Bool}, \mathsf{Char})$$
$$\quad \text{-- where } \mathit{frst} :: \mathsf{Lens}\ (\mathsf{Char}, \mathsf{Char})\ (\mathsf{Bool}, \mathsf{Char})\ \mathsf{Bool}\ \mathsf{Char}$$

Lens s t a b means:

- s contains an a
- replacing the a with a b changes the outer type from s to t

Why do we care?

- Lenses generalise to **optics**: a rich vocabulary of operations for data access and manipulation

Why do we care?

- Lenses generalise to **optics**: a rich vocabulary of operations for data access and manipulation
- Lenses **compose**:

$$(\circ) :: \mathsf{Lens}\ s\ t\ u\ v \rightarrow \mathsf{Lens}\ u\ v\ a\ b \rightarrow \mathsf{Lens}\ s\ t\ a\ b$$

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

Why do we care?

- Lenses generalise to **optics**: a rich vocabulary of operations for data access and manipulation
- Lenses **compose**:

$$(\circ) :: \mathrm{Lens}\ s\ t\ u\ v \to \mathrm{Lens}\ u\ v\ a\ b \to \mathrm{Lens}\ s\ t\ a\ b$$

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

- Lenses are **first-class values** (e.g. can be passed as arguments, stored in data structures, etc.)

Getters

A Getter s a is a function from s to a.

Type formation:

$$\mathsf{Getter}\ (s :: \mathsf{Type})\ (a :: \mathsf{Type}) :: \mathsf{Type}$$

Introduction:

$$\mathit{to} :: (s \to a) \to \mathsf{Getter}\ s\ a$$

Elimination:

$$\mathit{view}_G :: \mathsf{Getter}\ s\ a \to s \to a$$

Setters

A Setter $s\ t\ a\ b$ means replacing some as inside s with bs produces a t.

Type formation:

$$\mathrm{Setter}\ (s :: \mathrm{Type})\ (t :: \mathrm{Type})\ (a :: \mathrm{Type})\ (b :: \mathrm{Type}) :: \mathrm{Type}$$

Introduction:

$$\mathit{sets} :: ((a \to b) \to s \to t) \to \mathrm{Setter}\ s\ t\ a\ b$$

Elimination:

$$\mathit{over} :: \mathrm{Setter}\ s\ t\ a\ b \to (a \to b) \to s \to t$$

Laws:

$$\begin{aligned} \mathit{over}\ s\ \mathit{id} &\equiv \mathit{id} \\ \mathit{over}\ s\ f \,.\, \mathit{over}\ s\ g &\equiv \mathit{over}\ s\ (f \,.\, g) \end{aligned}$$

Subtyping?

$$
\begin{array}{l}
\mathit{frst}\ \ :: \mathsf{Lens}\ (a, x)\ (b, x)\ a\ b \\
\mathit{over} :: \mathsf{Setter}\ s\ t\ a\ b \to (a \to b) \to s \to t
\end{array}
$$

$$\mathit{lensToSetter} :: \mathsf{Lens}\ s\ t\ a\ b \to \mathsf{Setter}\ s\ t\ a\ b$$

$$\mathit{over}\ (\mathit{lensToSetter}\ \mathit{frst}) :: (a \to b) \to (a, x) \to (b, x)$$

- Every Lens gives a Getter and a Setter
- A Getter or a Setter alone doesn't give a Lens
- Could we say Lens is a subtype of Getter and of Setter? But Haskell doesn't have subtyping...
- Explicit conversions are painful

Subsumption

- How can we make *over frst* well-typed, without subtyping?
- The `lens` approach: use the van Laarhoven optic representation and rely on **subsumption**

$$\textbf{type}\ \mathrm{LensVL}\ \ s\ t\ a\ b = \textbf{forall}\ f\,.\ \mathrm{Functor}\ f \Rightarrow (a \to f\ b) \to s \to f\ t$$

$$\textbf{type}\ \mathrm{SetterVL}\ s\ t\ a\ b = \textbf{forall}\ f\,.\ \mathrm{Settable}\ f \Rightarrow (a \to f\ b) \to s \to f\ t$$

(Settable f says that f is Identity.)

Subsumption, the good

- It's remarkable that this works at all
- Lens is just a type synonym!
- Function composition (.) is optic composition
- Definitions are extremely polymorphic, so can be combined in unforeseen ways
- Varying the constraints and generalizing gives a whole zoo of different kinds of optic (including **profunctor optics**)

Subsumption, the bad

- Too transparent:

$$\mathit{frst} :: \mathsf{Functor}\ f \Rightarrow (a \to f\ a) \to (a, x) \to f\ (a, x)$$

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

Subsumption, the bad

- Too transparent:

$$\mathit{frst} :: \mathsf{Functor}\ f \Rightarrow (a \to f\ a) \to (a, x) \to f\ (a, x)$$

$$\mathit{frst}\ .\ \mathit{to}\ \mathit{not} :: (\mathsf{Contravariant}\ f, \mathsf{Functor}\ f) \Rightarrow (\mathsf{Bool} \to f\ \mathsf{Bool}) \to (\mathsf{Bool}, x) \to f\ (\mathsf{Bool}, x)$$

- Too expressive:

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

(Nothing can be both Contravariant and Settable.)

Well-Typed

Subsumption, the bad

- Too transparent:

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

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

- Too expressive:

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

  (Nothing can be both Contravariant and Settable.)

- Error messages are terrible.

The need for abstraction

The fundamental issue is a **lack of abstraction**:

- optic implementations are exposed as type synonyms (subsumption won't work otherwise)
- so we can't distinguish interfaces from implementations.

Can we design a library that:

- uses opaque abstractions for optics
- retains (some sort of) subtyping
- gives good type inference and error messages

Defining a family of optics

```
type OpticKind = Type

data A_Getter :: OpticKind
data A_Setter :: OpticKind
data A_Lens   :: OpticKind
```

Defining a family of optics

```
type OpticKind = Type

data A_Getter :: OpticKind
data A_Setter :: OpticKind
data A_Lens   :: OpticKind
```

```
newtype Optic (k :: OpticKind) (i :: IxList) s t a b = Optic {...}
type Optic' k i s a = Optic k i s s a a
```

Defining a family of optics

```
type OpticKind = Type

data A_Getter :: OpticKind
data A_Setter :: OpticKind
data A_Lens   :: OpticKind
```

```
newtype Optic (k :: OpticKind) (i :: IxList) s t a b = Optic {...}
type Optic' k i s a = Optic k i s s a a
```

```
type Getter s   a   = Optic' A_Getter NoIx s   a
type Setter s t a b = Optic  A_Setter NoIx s t a b
type Lens   s t a b = Optic  A_Lens   NoIx s t a b
```

Typeclass overloading for “subtyping”

```
class Is (k :: OpticKind) (l :: OpticKind) where ...

instance Is k        k         where ...
instance Is A_Lens A_Setter where ...
```

Typeclass overloading for “subtyping”

```
class Is (k :: OpticKind) (l :: OpticKind) where ...

instance Is k        k        where ...
instance Is A_Lens A_Setter where ...
```

```
castOptic :: Is src dst ⇒ Optic src i s t a b → Optic dst i s t a b
```

"Subtyping" in practice

$$\mathit{frst} :: \mathsf{Lens}\ (a, x)\ (b, x)\ a\ b$$
$$\mathit{over} :: \mathsf{Setter}\ s\ t\ a\ b \rightarrow (a \rightarrow b) \rightarrow s \rightarrow t$$

$$\mathit{over}\ (\mathit{castOptic}\ \mathit{frst}) :: (a \rightarrow b) \rightarrow (a, x) \rightarrow (b, x)$$

We don't want to write *castOptic* at every call site!

Instead, change the type of *over*:

$$\mathit{over} :: \mathsf{Is}\ k\ \mathsf{A\_Setter} \Rightarrow \mathsf{Optic}\ k\ i\ s\ t\ a\ b \rightarrow (a \rightarrow b) \rightarrow s \rightarrow t$$

Similarly, unify the $\mathit{view}_{G,L}$ functions we had earlier:

$$\mathit{view} :: \mathsf{Is}\ k\ \mathsf{A\_Getter} \Rightarrow \mathsf{Optic'}\ k\ i\ s\ a \rightarrow s \rightarrow a$$

Composition of optics

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?

```
frst % frst :: Lens ((a, x), y) ((b, x), y) a b
frst % to not :: Getter (Bool, x) Bool
sets map % to not   -- type error
```

Composition of optics

```
class JoinKinds k l m | k l → m

instance JoinKinds A_Lens   A_Lens   A_Lens
instance JoinKinds A_Lens   A_Getter A_Getter
instance JoinKinds A_Setter A_Lens   A_Setter
...
  -- no instance for JoinKinds A_Setter A_Getter m.
```

Composition of optics

```
class JoinKinds k l m | k l → m

instance JoinKinds A_Lens   A_Lens   A_Lens
instance JoinKinds A_Lens   A_Getter A_Getter
instance JoinKinds A_Setter A_Lens   A_Setter
 ...
  -- no instance for JoinKinds A_Setter A_Getter m.
```

```
(%) :: JoinKinds k l m
     ⇒ Optic k NoIx s t u v
     → Optic l NoIx u v a b
     → Optic m NoIx s t a b
```

We can't use (.) for optic composition. And that's good!

Type inference and errors

Inferred types for compositions are useful:

$\mathit{frst}\ \%\ \mathit{to}\ \mathit{not} :: \mathrm{Optic\ A\_Getter}\ (\mathrm{Bool}, x)\ (\mathrm{Bool}, x)\ \mathrm{Bool}\ \mathrm{Bool}$

Type inference and errors

Inferred types for compositions are useful:

$\mathit{frst} \,\%\, \mathit{to}\ \mathit{not} :: \mathrm{Optic\ A\_Getter}\ (\mathrm{Bool}, x)\ (\mathrm{Bool}, x)\ \mathrm{Bool}\ \mathrm{Bool}$

Incorrect compositions are rejected:

$\mathit{sets}\ \mathit{map} \,\%\, \mathit{to}\ \mathit{not}$

```
 * A_Setter cannot be composed with A_Getter
 * In the expression: sets map % to not
```

Notions of substructure

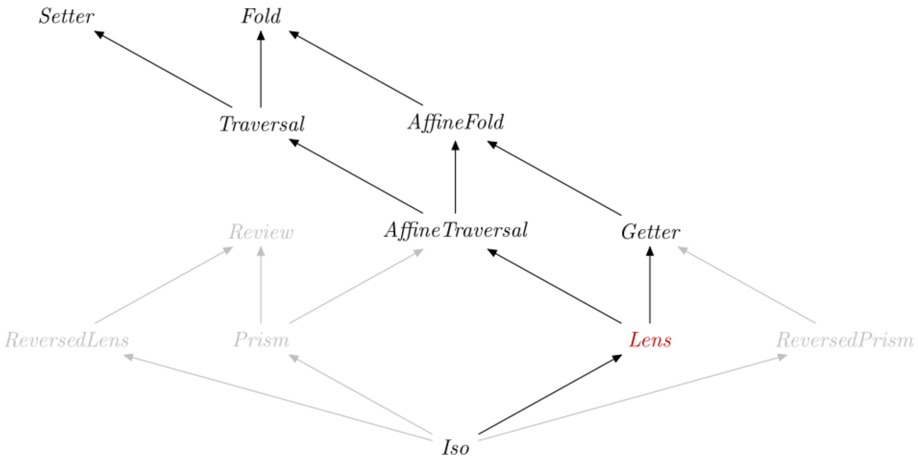
So far we've seen:

- Getter: s is a structure from which an a can be extracted
- Setter: replacing the as inside s with bs yields a t
- Lens: s contains an a field to be extracted or replaced

What if

- ...the substructure a occurs multiple times within s?
- ...a might not occur at all?
- ...s and a are the same structure?

The optic hierarchy



Well-Typed

Part II

Working with nested data

Records

```haskell
data Person = MkPerson { _personName :: String
                       , _personAge  :: Int
                       , _personPets :: [Pet] }

data Pet = MkPet { _petName :: String
                 , _petAge  :: Int }
```

Records

```
data Person = MkPerson { _personName :: String
                       , _personAge  :: Int
                       , _personPets :: [Pet] }

data Pet = MkPet { _petName :: String
                 , _petAge  :: Int }

getAges :: Person -> [Int]
getAges p = _personAge p : map _petAge (_personPets p)
```

Records

```haskell
data Person = MkPerson { _personName :: String
                       , _personAge  :: Int
                       , _personPets :: [Pet] }

data Pet = MkPet { _petName :: String
                 , _petAge  :: Int }

getAges :: Person -> [Int]
getAges p = _personAge p : map _petAge (_personPets p)

incAges :: Person -> Person
incAges p = p { _personAge  = _personAge p + 1
              , _personPets = map incPetAge (_personPets p) }
  where
    incPetAge t = t { _petAge = _petAge t + 1 }
```

Well-Typed

Lenses for Records

```haskell
data Person = MkPerson { _personName :: String
                       , _personAge  :: Int
                       , _personPets :: [Pet] }

data Pet = MkPet { _petName :: String
                 , _petAge  :: Int }
```

Well-Typed

Lenses for Records

```
data Person = MkPerson { _personName :: String
                       , _personAge  :: Int
                       , _personPets :: [Pet] }

data Pet = MkPet { _petName :: String
                 , _petAge  :: Int }
```

```
personName :: Lens' Person String
personName = lens _personName (λr v → r { _personName = v })
```

⋮

Lenses for Records

```
data Person = MkPerson { _personName :: String
                       , _personAge  :: Int
                       , _personPets :: [Pet] }

data Pet = MkPet { _petName :: String
                 , _petAge  :: Int }

$(makeLenses ''Person)   -- Gives personName :: Lens' Person String, ...
$(makeLenses ''Pet)      -- Gives petName :: Lens' Pet String, ...
```

Lenses for Records

```haskell
data Person = MkPerson { _personName :: String
                       , _personAge  :: Int
                       , _personPets :: [Pet] }

data Pet = MkPet { _petName :: String
                 , _petAge  :: Int }
```

```haskell
$(makeLenses ''Person)   -- Gives personName :: Lens' Person String, ...
$(makeLenses ''Pet)      -- Gives petName :: Lens' Pet String, ...
```

```haskell
getPersonName :: Person → String
getPersonName = view personName
```

What about *getAges*?

Folds

A Fold s a extracts a list of a elements from a structure s.[1]

Type formation:

$$\mathsf{Fold}\ (s :: \mathsf{Type})\ (a :: \mathsf{Type}) :: \mathsf{Type}$$

Introduction:

$$\mathit{folding}\ @[\,] :: (s \rightarrow [a]) \rightarrow \mathsf{Fold}\ s\ a$$

Elimination:

$$\mathit{toListOf} :: \mathsf{Is}\ k\ \mathsf{A\_Fold} \Rightarrow \mathsf{Optic'}\ k\ i\ s\ a \rightarrow s \rightarrow [a]$$

[1] Ignoring infinite structures.

Folds example

Some useful combinators:

$$
\begin{aligned}
&\mathit{folded} && :: \mathrm{Fold}\ [a]\ a \quad \text{-- actually more general, any Foldable} \\
&\mathit{summing} && :: \ (\mathrm{Is}\ k\ \mathrm{A\_Fold}, \mathrm{Is}\ l\ \mathrm{A\_Fold}) \\
& && \Rightarrow \mathrm{Optic'}\ k\ i\ s\ a \to \mathrm{Optic'}\ l\ j\ s\ a \to \mathrm{Fold}\ s\ a
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{ages} :: \mathrm{Fold\ Person\ Int} \\
&\mathit{ages} = \mathit{personAge}\ \text{`}\mathit{summing}\text{`}\ (\mathit{personPets}\ \%\ \mathit{folded}\ \%\ \mathit{petAge}) \\
&\mathit{getAges} :: \mathrm{Person} \to [\mathrm{Int}] \\
&\mathit{getAges} = \mathit{toListOf}\ \mathit{ages}
\end{aligned}
$$

What if we want to set as well as get?

Traversals

A Traversal $s\ t\ a\ b$ represents an s structure containing an ordered collection of as, which can be updated to bs yielding a t structure.

Type formation:

$$\mathsf{Traversal}\ (s :: \mathsf{Type})\ (t :: \mathsf{Type})\ (a :: \mathsf{Type})\ (b :: \mathsf{Type}) :: \mathsf{Type}$$

Introduction:

$$\begin{aligned} \mathit{traversalVL} \ &:: \ (\textbf{forall}\ f\ .\ \mathsf{Applicative}\ f \Rightarrow (a \to f\ b) \to s \to f\ t) \\ &\to \mathsf{Traversal}\ s\ t\ a\ b \end{aligned}$$

Elimination:

$$\begin{aligned} \mathit{traverseOf} \ &:: \ (\mathsf{Is}\ k\ \mathsf{A\_Traversal}, \mathsf{Applicative}\ f) \\ &\Rightarrow \mathsf{Optic}\ k\ i\ s\ t\ a\ b \to (a \to f\ b) \to s \to f\ t \end{aligned}$$

Laws: like the Setter laws, coming in a few slides

Traversals example

```
traversed :: Traversal [a] [b] a b    -- again, actually any Traversable
adjoin    :: (Is k A_Traversal, Is l A_Traversal)
          => Optic' k i s a → Optic' l j s a → Traversal' s a
```

```
ages :: Traversal' Person Int
ages = personAge `adjoin` (personPets % traversed % petAge)
getAges :: Person → [Int]
getAges = toListOf ages
incAges :: Person → Person
incAges = over ages (+1)
```

Defining traversals 1: lists

$$
\begin{aligned}
&\mathit{traverseList} :: \mathrm{Applicative}\ f \Rightarrow (a \to f\ b) \to [a] \to f\ [b]\\
&\mathit{traverseList}\ k\ [\,] \qquad\quad = \mathit{pure}\ [\,]\\
&\mathit{traverseList}\ k\ (x : xs) = (:) <\$> k\ x <*> \mathit{traverseList}\ k\ xs
\end{aligned}
$$

Defining traversals 1: lists

$$
\begin{array}{l}
\mathit{traverseList} :: \mathsf{Applicative}\ f \Rightarrow (a \to f\ b) \to [a] \to f\ [b] \\
\mathit{traverseList}\ k\ []\qquad\quad = \mathit{pure}\ [] \\
\mathit{traverseList}\ k\ (x : xs) = (:)\ {<}\${>}\ k\ x\ {<}{*}{>}\ \mathit{traverseList}\ k\ xs
\end{array}
$$

$$
\begin{array}{l}
\mathit{listTraversal} :: \mathsf{Traversal}\ [a]\ [b]\ a\ b \\
\mathit{listTraversal} = \mathit{traversalVL}\ \mathit{traverseList}
\end{array}
$$

Defining traversals 1: lists

$$
\begin{aligned}
&\mathit{traverseList} :: \mathrm{Applicative}\ f \Rightarrow (a \to f\ b) \to [a] \to f\ [b] \\
&\mathit{traverseList}\ k\ [\,] \qquad\quad = \mathit{pure}\ [\,] \\
&\mathit{traverseList}\ k\ (x : xs) = (:) <\$> k\ x <*> \mathit{traverseList}\ k\ xs
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{listTraversal} :: \mathrm{Traversal}\ [a]\ [b]\ a\ b \\
&\mathit{listTraversal} = \mathit{traversalVL}\ \mathit{traverseList}
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{class}\ (\mathrm{Functor}\ t, \mathrm{Foldable}\ t) \Rightarrow \mathrm{Traversable}\ t\ \mathbf{where} \\
&\quad \mathit{traverse} :: \mathrm{Applicative}\ f \Rightarrow (a \to f\ b) \to t\ a \to f\ (t\ b)
\end{aligned}
$$

Defining traversals 2: pairs

$$\mathit{traversePair} :: \mathsf{Applicative}\ f \Rightarrow (a \rightarrow f\ b) \rightarrow (a, a) \rightarrow f\ (b, b)$$
$$\mathit{traversePair}\ k\ (x, y) = (,)\ {<}\${>}\ k\ x\ {<}{*}{>}\ k\ y$$

$$\mathit{pairTraversal} :: \mathsf{Traversal}\ (a, a)\ (b, b)\ a\ b$$
$$\mathit{pairTraversal} = \mathit{traversalVL}\ \mathit{traversePair}$$

Not the same as the Traversable $((,)\ a)$ instance!

Using traversals

Every Traversal is a Fold and a Setter, so we can use *toListOf*, *over*, ...

We can't use *view* (in `optics`) because a Traversal isn't a Getter.

Using traversals

Every Traversal is a Fold and a Setter, so we can use *toListOf*, *over*, ...

We can't use *view* (in `optics`) because a Traversal isn't a Getter.

In more interesting situations we can use *traverseOf*:

$$
\begin{aligned}
\mathit{traverseOf} \; &:: \; (\mathrm{Is}\; k\; \mathrm{A\_Traversal}, \mathrm{Applicative}\; f) \\
&\Rightarrow \mathrm{Optic}\; k\; i\; s\; t\; a\; b \to (a \to f\; b) \to s \to f\; t
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{eg} :: (\mathrm{String}, \mathrm{String}) \to \mathrm{IO}\; (\mathrm{String}, \mathrm{String}) \\
&\mathit{eg} = \mathit{traverseOf}\; \mathit{pairTraversal}\; (\lambda s \to \mathit{putStrLn}\; s \gg \mathit{getLine})
\end{aligned}
$$

What's not a traversal?

$$
\begin{aligned}
&\mathit{traverseOf}\ o\ \mathit{pure} \equiv \mathit{pure}\\
&\mathit{fmap}\ (\mathit{traverseOf}\ o\ f)\ .\ \mathit{traverseOf}\ o\ g\\
&\quad \equiv \mathit{getCompose}\ .\ \mathit{traverseOf}\ o\ (\mathsf{Compose}\ .\ \mathit{fmap}\ f\ .\ g)
\end{aligned}
$$

The Traversal laws generalize the laws for Setters.

They imply that:

- traversing does not change the number of substructures
- a traversal does not visit the same position more than once

Thus there's no traversal visiting the elements of a Set.

From traversals to lenses

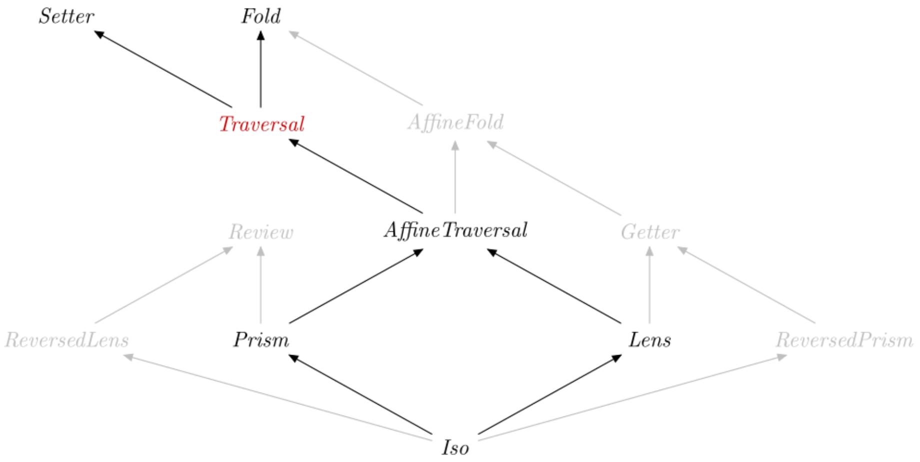
$$
\begin{aligned}
&\textbf{type}\ \mathsf{TraversalVL}\ s\ t\ a\ b = \textbf{forall}\ f\ .\ \mathsf{Applicative}\ f \Rightarrow \\
&\qquad (a \rightarrow f\ b) \rightarrow s \rightarrow f\ t \\
&\textbf{type}\ \mathsf{LensVL}\ s\ t\ a\ b = \textbf{forall}\ f\ .\ \mathsf{Functor}\ f \Rightarrow \\
&\qquad (a \rightarrow f\ b) \rightarrow s \rightarrow f\ t
\end{aligned}
$$

From traversals to lenses

$$
\begin{aligned}
&\textbf{type}\ \mathsf{TraversalVL}\ s\ t\ a\ b = \textbf{forall}\ f\ .\ \mathsf{Applicative}\ f \Rightarrow \\
&\qquad\qquad (a \to f\ b) \to s \to f\ t \\
&\textbf{type}\ \mathsf{LensVL}\ \ s\ t\ a\ b = \textbf{forall}\ f\ .\ \mathsf{Functor}\ f \Rightarrow \\
&\qquad\qquad (a \to f\ b) \to s \to f\ t
\end{aligned}
$$

- Both lift effectful operations from inner values to outer structures
- Lenses can use only *fmap* from Functor
- Traversals also have access to *pure* and $<\!*\!>$ from Applicative
- Call site can choose an f (e.g. Identity to get *over*)

The optic hierarchy, again



Well-Typed

More notions of substructure

We've seen:

- Lens: both a Getter ($s \to a$) and a Setter ($s \to b \to t$)
- Fold: s is a structure from which many as can be extracted
- Traversal: s contains as that can be iterated over

We've not (yet) covered:

- AffineFold and AffineTraversal: s contains at most one a
- Prism: s may be constructed from an a or from something else
- Iso: s is convertible to and from a

Part III

Duplicate Record Fields

Redefining record fields with `DuplicateRecordFields`

Normally, each record field in a module must have a distinct field label. `DuplicateRecordFields` lifts this restriction.

```
data Person = MkPerson { name :: String
                       , age  :: Integer
                       , pets :: [Pet] }
data Pet = MkPet { name :: String
                 , age  :: Int }
```

Binding *name* at the top level is not allowed yet, but will be with `NoFieldSelectors` in GHC 9.2.

Using record fields

- Construction:

$$\mathit{alice} = \mathsf{MkPerson}\ \{\mathit{name} = \texttt{"Alice"}, \mathit{age} = 65, \mathit{pets} = [\,]\}$$

Using record fields

- Construction:

```
alice = MkPerson {name = "Alice", age = 65, pets = []}
```

- Pattern matching:

```
hasPets (MkPerson {pets = xs}) = not (null xs)
```

Using record fields

- Construction:

$alice = \text{MkPerson}\ \{name = \texttt{"Alice"}, age = 65, pets = [\,]\}$

- Pattern matching:

$hasPets\ (\text{MkPerson}\ \{pets = xs\}) = not\ (null\ xs)$

- Record update:

$init\ p = p\ \{age = 0, pets = [\,]\}$

Using record fields

- Construction:

$$alice = \mathrm{MkPerson}\ \{name = \texttt{"Alice"}, age = 65, pets = [\,]\}$$

- Pattern matching:

$$hasPets\ (\mathrm{MkPerson}\ \{pets = xs\}) = not\ (null\ xs)$$

- Record update:

$$init\ p = p\ \{age = 0, pets = [\,]\}$$

- Selector functions:

$$hasPets\ p = null\ (pets\ p)$$

Ambiguous selector functions

We could use *pets* unambiguously, but what about *name*?

```
isAlice p = name p == "Alice"
```

Ambiguous selector functions

We could use *pets* unambiguously, but what about *name*?

```
isAlice p = name p == "Alice"
```

`DuplicateRecordFields` made limited use of type information to disambiguate selectors, but it is going away from GHC 9.2 onwards.

What to do instead?

Ambiguous selector functions

We could use *pets* unambiguously, but what about *name*?

```
isAlice p = name p == "Alice"
```

`DuplicateRecordFields` made limited use of type information to disambiguate selectors, but it is going away from GHC 9.2 onwards.

What to do instead?

Don't use clashing names?

Option 1: Using the module system

Import each selector qualified with a type-specific prefix.

```haskell
import qualified MyModule (Person (..)) as Person
isAlice p = Person.name p == "Alice"
```

Well-Typed

Option 1: Using the module system

Import each selector qualified with a type-specific prefix.

```
import qualified MyModule (Person (..)) as Person
isAlice p = Person.name p == "Alice"
```

- Doesn't need any extensions
- Simple, no fancy types
- Relatively verbose
- Doesn't work in the defining module

Option 2: Record dot syntax

GHC 9.2 will support `OverloadedRecordDot`, which uses special syntax for field selection with typeclass-based name resolution.

$(.\mathit{name}) :: \mathrm{HasField}\ \texttt{"name"}\ r\ a \Rightarrow r \rightarrow a$

Option 2: Record dot syntax

GHC 9.2 will support `OverloadedRecordDot`, which uses special syntax for field selection with typeclass-based name resolution.

$$(.\mathit{name}) :: \mathsf{HasField}\ \texttt{"name"}\ r\ a \Rightarrow r \to a$$

$$\begin{aligned}&\mathit{isAlice} :: \mathsf{Person} \to \mathsf{Bool}\\&\mathit{isAlice}\ p = p.\mathit{name} == \texttt{"Alice"}\end{aligned}$$

Option 2: Record dot syntax

GHC 9.2 will support `OverloadedRecordDot`, which uses special syntax for field selection with typeclass-based name resolution.

$(.\mathit{name}) :: \mathsf{HasField}\ \texttt{"name"}\ r\ a \Rightarrow r \to a$

$\mathit{isAlice} :: \mathsf{Person} \to \mathsf{Bool}$
$\mathit{isAlice}\ p = p.\mathit{name} == \texttt{"Alice"}$

- Brand new extension and new syntax (for Haskell)
- HasField class magically solved by GHC
- Updates not really supported in GHC 9.2, should be in the future
- Not as compositional as lenses

Option 3: `OverloadedLabels` + `optics`

The meaning of an overloaded label *#name* depends on the text of the label and its type.

$\#\mathit{name} :: \mathsf{IsLabel}\ \texttt{"name"}\ t \Rightarrow t$

Option 3: `OverloadedLabels` + `optics`

The meaning of an overloaded label *#name* depends on the text of the label and its type.

```
#name :: IsLabel "name" t => t
```

```
isAlice :: Person -> Bool
isAlice p = view #name p == "Alice"
```

Well-Typed

Option 3: `OverloadedLabels` + `optics`

The meaning of an overloaded label *#name* depends on the text of the label and its type.

$$\#\mathit{name} :: \mathsf{IsLabel}\ \texttt{"name"}\ t \Rightarrow t$$

$$\begin{aligned}&\mathit{isAlice} :: \mathsf{Person} \rightarrow \mathsf{Bool}\\&\mathit{isAlice}\ p = \mathit{view}\ \#\mathit{name}\ p == \texttt{"Alice"}\end{aligned}$$

- Provides lenses, not just selectors, so fits with other optics
- `OverloadedLabels` is used differently by different libraries
- Syntax can be clunky

OverloadedLabels example

```haskell
{-# LANGUAGE DuplicateRecordFields, OverloadedLabels #-}

data Person = MkPerson { name :: String
                       , age  :: Int
                       , pets :: [Pet] } deriving Generic

data Pet = MkPet { name :: String
                 , age  :: Int } deriving Generic
```

OverloadedLabels example

```haskell
{-# LANGUAGE DuplicateRecordFields, OverloadedLabels #-}

data Person = MkPerson { name :: String
                       , age  :: Int
                       , pets :: [Pet] } deriving Generic

data Pet = MkPet { name :: String
                 , age  :: Int } deriving Generic
```

```haskell
getPersonName :: Person -> String
getPersonName = view #name

ages :: Traversal' Person Int
ages = #age `adjoin` (#pets % traversed % #age)
```

The `HasField` class

$$\textbf{class}\ \mathrm{HasField}\ x\ r\ a \mid x\ r \to a\ \textbf{where}$$
$$\quad getField :: r \to a$$

`OverloadedRecordDot` syntax desugaring:

$$p.name \mapsto getField\ @\text{"name"}\ p :: \mathrm{HasField}\ \text{"name"}\ r\ a \Rightarrow r \to a$$

GHC will automatically solve constraints like

HasField "name" Person String

when *name* is a field of Person

The `IsLabel` class

$$\textbf{class}\ \mathsf{IsLabel}\ (n :: \mathsf{Symbol})\ t\ \textbf{where}$$
$$\quad \mathit{fromLabel} :: t$$

`OverloadedLabels` syntax desugaring:

$$\#\mathit{name} \mapsto \mathit{fromLabel}\ \texttt{@"name"} :: \mathsf{IsLabel}\ \texttt{"name"}\ t \Rightarrow t$$

Well-Typed

The `IsLabel` class

```
class IsLabel (n :: Symbol) t where
  fromLabel :: t
```

`OverloadedLabels` syntax desugaring:

```
#name ↦ fromLabel @"name" :: IsLabel "name" t ⇒ t
```

```
class LabelOptic (n :: Symbol) k s t a b where
  labelOptic :: Optic k NoIx s t a b

instance LabelOptic n k s t a b ⇒ IsLabel n (Optic k NoIx s t a b)
```

Operator soup (with overloaded labels)

$$\begin{aligned}
&\mathit{getPersonName} :: \mathsf{Person} \to \mathsf{String} \\
&\mathit{getPersonName}\ p = p\ \hat{}.\ \#\mathit{name} \\
&\mathit{getPetNames} :: \mathsf{Person} \to [\mathsf{String}] \\
&\mathit{getPetNames}\ p = p\ \hat{}..\ \#\mathit{pets}\ \%\ \mathit{traversed}\ \%\ \#\mathit{name} \\
&\mathit{incPetAges} :: \mathsf{Person} \to \mathsf{Person} \\
&\mathit{incPetAges}\ p = p\ \&\ \#\mathit{pets}\ \%\ \mathit{traversed}\ \%\ \#\mathit{age}\ \%{\sim}\ (+1)
\end{aligned}$$

Operator soup (without overloaded labels)

$$
\begin{aligned}
&\mathit{getPersonName} :: \mathsf{Person} \rightarrow \mathsf{String} \\
&\mathit{getPersonName}\ p = p\ \hat{}\,.\ \mathit{personName} \\
&\mathit{getPetNames} :: \mathsf{Person} \rightarrow [\mathsf{String}] \\
&\mathit{getPetNames}\ p = p\ \hat{}\,..\ \mathit{personPets}\ \%\ \mathit{traversed}\ \%\ \mathit{petName} \\
&\mathit{incPetAges} :: \mathsf{Person} \rightarrow \mathsf{Person} \\
&\mathit{incPetAges}\ p = p\ \&\ \mathit{personPets}\ \%\ \mathit{traversed}\ \%\ \mathit{petAge}\ \%\!\sim (+1)
\end{aligned}
$$

Most common operators

$(\hat{}.)$	*view*	$s \to \text{Getter}\ s\ a \to a$
$(\hat{}..)$	*toListOf*	$s \to \text{Fold}\ s\ a \to [a]$
$(.\sim)$	*set*	$\text{Setter}\ s\ t\ a\ b \to b \to s \to t$
$(\%\sim)$	*over*	$\text{Setter}\ s\ t\ a\ b \to (a \to b) \to s \to t$
$(\&)$	*flip* (\$)	$s \to (s \to t) \to t$

Conclusions so far

We have seen:

- Interfaces of some key optics
- How `optics` captures those interfaces explicitly
- How `optics` makes working with records more convenient

Coming up next:

- abstracting some general lessons about library design in Haskell

Part IV

Reflections on library design

Identify your interfaces

We've repeatedly seen this pattern for defining interfaces:

Type formation	Lens *s t a b*
Introduction	*lens*
Elimination	*view*, *set*
Combinators	(%), *adjoin*, ...
Laws	GetPut, PutGet, PutPut

What are the key abstractions in your library? Are they documented?

Pulling a Monoid out of a hat

Types express **meaning**, not just **structure**.

In `optics`, *view* requires the optic to be a Getter.

In `lens`, *view* works for Fold:

```
view folded :: (MonadReader (f a) m, Foldable f, Monoid a) ⇒ m a

view folded [Just "abc", Nothing, Just "def"]
Just "abcdef"

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

Implementation hiding

- Use the module system to hide implementation details
- `.Internal` modules are okay, if you have a clear public API
- Consider exposing smart constructors and lenses instead of data constructors and record fields

Typeclasses are for overloading, not for abstraction

- Typeclasses are useful when you need **overloading** (ToJSON)
- . . . or for **global coherence** (Ord)
- . . . but not just to introduce an interface!

Design for type inference

- A little polymorphism can reduce noise, making code simpler
- But too much risks confusion:
  - Will the compiler be able to algorithmically infer appropriate types?
  - Will future readers of your code be able to mentally infer them?
- In `optics`:
  - Introduction forms (e.g. *lens*) return a concrete optic kind
  - Elimination forms (e.g. *view*, *set*) are polymorphic (Is)
  - Composition infers the result optic kind from the arguments

The dangers of excessive (class) polymorphism

- Confusing for users trying to understand what library API means
- Introduces risk of **ambiguity** (the *show* . *read* problem)
- Sometimes unexpected things typecheck

Performance concerns of polymorphism

- Passing class dictionaries at runtime can be bad for performance
- Specialization and inlining can get rid of this overhead, but only if
  - you're lucky
  - definitions are small enough to be inlind automatically, or
  - you use `{-# SPECIALIZE #-}` and `{-# INLINE #-}` pragmas correctly

Theorems for free?

- Polymorphism does sometimes make it possible to tell something about the function just from its type, via **parametricity**
- $\mathit{foo} :: \mathbf{forall}\ a\ .\ a \to a \to a$ must either diverge or return one of its arguments
- Useful for very general combinators; often less useful once class constraints get involved
- Sometimes helpful with higher-rank types to impose restrictions on the caller (e.g. the ST monad).

Well-Typed

Antipattern: bottom-up propagation of constraints

MyApp:

```haskell
doStuff :: (MonadIO m, MonadReader Int m) => m ()
doStuff = liftIO . print =<< ask
```

MyApp2:

```haskell
doMoreStuff :: (MonadIO m, MonadReader Int m
               , MonadWriter (Maybe String) m) => m ()
doMoreStuff = doStuff >> tell (Just "Hello")
```

Main:

```haskell
app :: MonadIO m => m (Maybe String)
app = execWriterT (runReaderT doMoreStuff 42)
```

Well-Typed

Prefer top-down design

- When starting out, **identify interfaces** between components
- Write down **type signatures**
- **Push requirements down**, rather than bubbling constraints up
- Hide implementation details
- Clearer for readers, easier to modify, and better optimized

Thanks for listening

#optics channel on Discord

`https://github.com/well-typed/optics-zurihac-2021`

`https://hackage.haskell.org/package/optics`

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

Part V

Appendix: optics libraries

Dependency structure

One "batteries-included" `optics` package for applications, smaller packages for libraries

- `optics-core`
- `optics-extra`
- `optics-th`
- `optics-vl`

- `template-haskell-optics`
- lots of other `*-optics` packages

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
- Easy to convert between `optics` and `lens` representations

Optics libraries

- `lens`: de facto standard, van Laarhoven representation, large dependency footprint (2012)
- `microlens`: family of `lens`-compatible packages, very few dependencies (2015)
- `optics`: provides an abstract interface, relatively few dependencies (2019)
- `profunctor-optics`: profunctor representation without the newtype wrapper (2019)
- ...

Evaluation

Advantages of `optics`:

- Relatively good type errors and simple inferred types
- Well-documented, selective API
- Function and lens composition clearly distinguished: `( . )` vs `( % )`

Advantages of `lens` and friends:

- Can provide (some) optics without dependencies besides base
- More featureful APIs available
- More polymorphism; many things "just work"
- `( . )` for lens composition is neat

Part VI

Appendix: More optic kinds

Prisms

A Prism $s\ t\ a\ b$ has a **constructor** and a **matcher**. They are mostly useful as traversals selecting one constructor from a sum datatype.

Type formation:

$$\mathrm{Prism}\ (s :: \mathrm{Type})\ (t :: \mathrm{Type})\ (a :: \mathrm{Type})\ (b :: \mathrm{Type}) :: \mathrm{Type}$$

Introduction:

$$\mathit{prism} :: (b \to t) \to (s \to \mathrm{Either}\ t\ a) \to \mathrm{Prism}\ s\ t\ a\ b$$

Elimination:

$$\begin{array}{lll} \mathit{review} & :: \mathrm{Is}\ k\ \mathrm{A\_Review} & \Rightarrow \mathrm{Optic'}\ k\ i\ t\ b \to b \to t \\ \mathit{matching} & :: \mathrm{Is}\ k\ \mathrm{An\_AffineTraversal} & \Rightarrow \mathrm{Optic}\ k\ i\ s\ t\ a\ b \to s \to \mathrm{Either}\ t\ a \end{array}$$

Isos

An $\mathrm{Iso}\ s\ t\ a\ b$ is an **isomorphism**, i.e. the types are interconvertible.

Type formation:

$$\mathrm{Iso}\ (s :: \mathrm{Type})\ (t :: \mathrm{Type})\ (a :: \mathrm{Type})\ (b :: \mathrm{Type}) :: \mathrm{Type}$$

Introduction:

$$\mathit{iso} :: (s \to a) \to (b \to t) \to \mathrm{Iso}\ s\ t\ a\ b$$

Elimination:

$$\mathit{view} :: \mathrm{Is}\ k\ \mathrm{A\_Getter} \Rightarrow \mathrm{Optic'}\ k\ i\ s\ a \to s \to a$$
$$\mathit{review} :: \mathrm{Is}\ k\ \mathrm{A\_Review} \Rightarrow \mathrm{Optic'}\ k\ i\ t\ b \to b \to t$$

Laws: *view* and *review* must be inverses.

Indexed Lenses

An IxLens $i\ s\ t\ a\ b$ augments a Lens with an index value.

Type formation:

$$\text{IxLens}\ (s :: \text{Type})\ (t :: \text{Type})\ (a :: \text{Type})\ (b :: \text{Type}) :: \text{Type}$$

Introduction:

$$\mathit{ilens} :: (s \to (i, a)) \to (s \to b \to t) \to \text{IxLens}\ i\ s\ t\ a\ b$$

Elimination:

$$\mathit{iview} :: \text{Is}\ k\ \text{A\_Getter} \Rightarrow \text{Optic'}\ k\ \text{'}[i]\ s\ a \to s \to (i, a)$$

$$\mathit{iset} :: \text{Is}\ k\ \text{A\_Setter} \Rightarrow \text{Optic}\ k\ \text{'}[i]\ s\ t\ a\ b \to (i \to b) \to s \to t$$

Profunctor optics

The **van Laarhoven** representation of lenses is

$$\textbf{type}\ \mathrm{LensVL}\ s\ t\ a\ b = \textbf{forall}\ f\ .\ \mathrm{Functor}\ f \Rightarrow (a \to f\ b) \to s \to f\ t$$

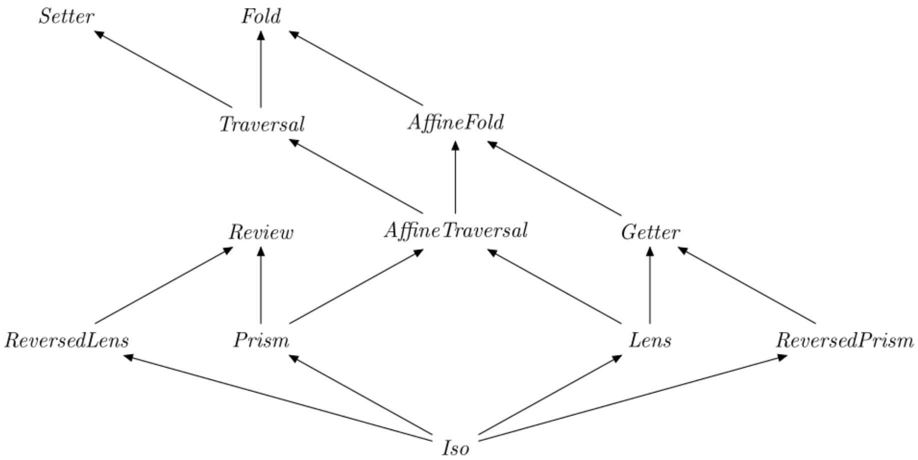
The **profunctor** representation is

$$\textbf{type}\ \mathrm{LensP}\ s\ t\ a\ b = \textbf{forall}\ p\ .\ \mathrm{Strong}\ p \Rightarrow p\ a\ b \to p\ s\ t$$

$$\textbf{class}\ \mathrm{Profunctor}\ p \Rightarrow \mathrm{Strong}\ p\ \textbf{where} \ldots$$

- Profunctor representation is theoretically cleaner; all optics are "profunctor transformers" for suitably-constrained profunctors
- If working with representations directly, the van Laarhoven representation is practically easier because it reuses more of base
- `optics` uses (indexed) profunctors internally

The optic hierarchy



Well-Typed