Lenses: viewing and updating data structures in Haskell

Twan van Laarhoven
twanvl@gmail.com
http://twanvl.nl/

Institute for Computing and Information Sciences – Intelligent Systems
Radboud University Nijmegen

Foundations Seminar
17th May 2011

Outline

What are lenses?

Practicalities

Implementations

Plates

What are lenses?

Motivating example

```
data Person = Person { name :: String, age :: Int }

-- Retrieving a value
name :: Person → String
name twan == "Twan van Laarhoven"

-- Updating a field
twan { age = age twan + 1 }    -- ugly

-- Even worse with nested fields
paper { author =
    (author paper) { age = age (author paper) + 1 } }
```

IN·DEI·NOMINE

Motivating example

```
data Person = Person { name :: String, age :: Int }

-- Retrieving a value
name :: Person → String
name twan == "Twan van Laarhoven"

-- Updating a field
twan { age = age twan + 1 }    -- ugly

-- Even worse with nested fields
paper { author =
    (author paper) { age = age (author paper) + 1 } }
```

Getters and setters

```
data Person = Person { name :: String, age :: Int }

-- Declare setter functions
setName :: String → Person → Person
setName n p = p { name = n }
setAge :: Int → Person → Person
setAge a p = p { age = a }

-- Declare modify functions
modifyName :: (String → String) → Person → Person
modifyName f p = p { name = f (name p) }
```

Introducing lenses

— Packing it up in a data type

data Lens α β
get :: Lens $\alpha\ \beta \to \alpha \to \beta$
set :: Lens $\alpha\ \beta \to \beta \to \alpha \to \alpha$
modify :: Lens $\alpha\ \beta \to (\beta \to \beta) \to \alpha \to \alpha$

— The example now becomes

modify author (modify age (+1)) paper
  where author :: Lens Paper Person
    age :: Lens Person Int

Laws

— Either modify or set will do

$\text{modify}\ \ell\ f\ a = \text{set}\ \ell\ (f\ (\text{get}\ \ell\ a))\ a$

$\text{set}\ \ell\ b = \text{modify}\ \ell\ (\text{const}\ b)$

— get/set laws

$\text{set}\ \ell\ (\text{get}\ \ell\ a)\ a = a$

$\text{get}\ \ell\ (\text{set}\ \ell\ b\ a) = b$

$\text{set}\ \ell\ c\ (\text{set}\ \ell\ b\ a) = \text{set}\ \ell\ c\ a$

— Or in terms of modify

$\text{modify}\ \ell\ \text{id} = \text{id}$

$\text{modify}\ \ell\ f \circ \text{modify}\ \ell\ g = \text{modify}\ \ell\ (f \circ g)$

$\text{get}\ \ell \circ \text{modify}\ \ell\ f = f \circ \text{get}\ \ell$

$\text{modify}\ \ell\ (\text{const}\ (\text{get}\ \ell\ a))\ a = a$

Example lenses

```
— Some lenses for tuples
fst :: Lens (α,β) α
snd :: Lens (α,β) β

— Some lenses for lists (not total!)
head :: Lens [α] α
tail :: Lens [α] [α]
at :: Int → Lens [α] α

— More examples
div :: Int → Lens Int Int
lookup :: α → Lens (Map α β) (Maybe β)
lines :: Lens String [String] — more on this later
```

Combinators

— Combinators

id :: Lens α α

($\circ$) :: Lens β γ $\rightarrow$ Lens α β $\rightarrow$ Lens α γ

pair :: Lens α β $\rightarrow$ Lens γ δ $\rightarrow$ Lens (α,γ) (β,δ)

unit :: Lens α ()

either :: Lens α β $\rightarrow$ Lens γ β $\rightarrow$ Lens (Either α γ) β

— The example now becomes

modify (age $\circ$ author) (+1) paper

Failed combinators

$\text{pair}' :: \text{Lens}\ \alpha\ \beta \rightarrow \text{Lens}\ \alpha\ \delta \rightarrow \text{Lens}\ \alpha\ (\beta,\delta)$

$\text{either}' :: \text{Lens}\ \alpha\ \beta \rightarrow \text{Lens}\ \gamma\ \delta \rightarrow \text{Lens}\ (\text{Either}\ \alpha\ \gamma)\ (\text{Either}\ \beta\ \delta)$

$\text{const} :: \beta \rightarrow \text{Lens}\ \alpha\ \beta$

IN·DEI·NOMINE

Practicalities

Bijections

Bijections live between lenses and functions.

data Iso α β
apply :: Iso α $\beta \to \alpha \to \beta$
inverse :: Iso α $\beta \to$ Iso β α

— *Laws*
inverse (inverse i) = i
apply (inverse i) ∘ apply i = apply i ∘ apply (inverse i) = id

— *Examples*
(+) :: Integer → Iso Integer Integer
lines :: Iso String [String]

Bijections

Bijections live between lenses and functions.

data Iso α β = Iso $(\alpha \to \beta)$ $(\beta \to \alpha)$
apply :: Iso α $\beta \to \alpha \to \beta$
inverse :: Iso α $\beta \to$ Iso β α

— *Laws*
inverse (inverse *i*) = *i*
apply (inverse *i*) $\circ$ apply *i* = apply *i* $\circ$ apply (inverse *i*) = id

— *Examples*
$(+)$ :: Integer $\to$ Iso Integer Integer
lines :: Iso String [String]

Overloading

$\textbf{class}$ Category $(\rightsquigarrow)$ $\textbf{where}$

  id :: $\alpha \rightsquigarrow \alpha$

  $(\circ)$ :: $(\beta \rightsquigarrow \gamma) \rightarrow (\alpha \rightsquigarrow \beta) \rightarrow (\alpha \rightsquigarrow \gamma)$

  — *Law:* $f \circ id = id \circ f = f$

  — *Law:* $f \circ (g \circ h) = (f \circ g) \circ h$

$\textbf{class}$ Category $(\rightsquigarrow) \Rightarrow$ IsoCategory $(\rightsquigarrow)$ $\textbf{where}$

  liftIso :: (Iso α β) $\rightarrow (\alpha \rightsquigarrow \beta)$

$\textbf{class}$ IsoCategory $(\rightsquigarrow) \Rightarrow$ LensCategory $(\rightsquigarrow)$ $\textbf{where}$

  liftLens :: (Lens α β) $\rightarrow (\alpha \rightsquigarrow \beta)$

$\textbf{class}$ LensCategory $(\rightsquigarrow) \Rightarrow$ Arrow $(\rightsquigarrow)$ $\textbf{where}$

  lift :: $(\alpha \rightarrow \beta) \rightarrow (\alpha \rightsquigarrow \beta)$

Automatic definition

```
-- Some record type
data Person = Person { name_ :: String, age_ :: Int }

-- Magic incantation
deriveLenses ''Person

-- This declares
name :: LensCategory (~>) => Person ~> String
age  :: LensCategory (~>) => Person ~> Int
```

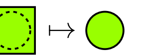
Implementations

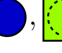
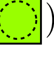
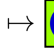
Implementations

Several implementations possible:

get/set	$(\alpha \to \beta, \beta \to \alpha \to \alpha)$
get/modify	$(\alpha \to \beta, (\beta \to \beta) \to (\alpha \to \alpha))$
store comonad	$\alpha \to (\beta, \beta \to \alpha)$
context	$\alpha \to (\beta, \partial\alpha/\partial\beta)$
residual	$\exists r. (\alpha \to (\beta, r)), (\beta, r) \to \alpha)$
transformer	$\forall f. \mathrm{Functor}\ f \Rightarrow (\beta \to f\beta) \to (\alpha \to f\alpha)$
	— *Functor f* $= \forall\gamma\delta. (\gamma \to \delta) \to (f\gamma \to f\delta)$

Simple lenses

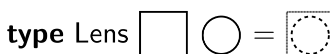
type Lens 

get/set lens   

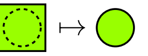
get/modify lens

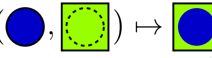
store comonad lens

Simple lenses

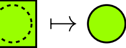


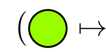
get/set lens

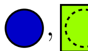




get/modify lens

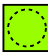
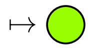
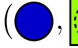


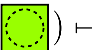
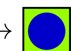


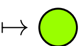
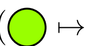
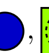


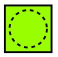
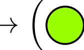
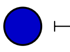
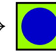
store comonad lens

Simple lenses

type Lens   = 

get/set lens  

get/modify lens   

store comonad lens    

Store comonad lenses

type SLens α β = $\alpha \rightarrow$ Store α β
data Store α β = Store { pos :: β, peek :: $\beta \rightarrow \alpha$ }

extract :: Store α $\beta \rightarrow \alpha$
duplicate :: Store α $\beta \rightarrow$ Store (Store α β) β

get ℓ a = pos (ℓ a)
set ℓ a = flip (peek (ℓ a))

Context lenses

— Definition

type CLens α β = $\alpha \to (\beta, \partial\alpha/\partial\beta)$

where $\partial\alpha/\partial\beta$ = "one hole context" = α with one β removed.
c.f. Huet's zipper.

Examples:

$\partial(\alpha,\beta)/\partial\alpha = \beta$

$\partial(\alpha,\alpha)/\partial\alpha =$ Either α α

$\partial[\alpha]/\partial\alpha = ([\alpha],[\alpha])$

but what is ∂Int/∂Int?

Context lenses

— Definition

type CLens α $\beta = \alpha \to (\beta, \partial\alpha/\partial\beta)$

where $\partial\alpha/\partial\beta =$ "one hole context" $= \alpha$ with one β removed.
c.f. Huet's zipper.

Examples:

$$\partial(\alpha,\beta)/\partial\alpha = \beta$$
$$\partial(\alpha,\alpha)/\partial\alpha = \text{Either } \alpha\ \alpha$$
$$\partial[\alpha]/\partial\alpha = ([\alpha],[\alpha])$$

but what is $\partial\text{Int}/\partial\text{Int}$?

IN·DEI·NOMINE

Residual lenses

— *Definition*

$$\textbf{type}\ \text{RLens}\ \alpha\ \beta = \exists r.\,(\alpha \to (\beta, r)), (\beta, r) \to \alpha)$$

$\text{fst} :: \text{RLens}\ (\alpha,\beta)\ \alpha$

$\text{fst} = \text{Iso}\ (\lambda(a,b) \to (a,b))\ (\lambda(a,b) \to (a,b))$

$\text{div} :: \text{Int} \to \text{RLens Int Int}$

$\text{div}\ d = \text{Iso}\ (\text{divMod}\ d)\ (\lambda(f,r) \to f * d + r)$

$r = \alpha$	get/set lens
$r = \beta \to \alpha$	store comanad lens
$r = \partial\alpha/\partial\beta$	context lens

Residual lenses

— *Definition*

type RLens α β = $\exists r.$ Iso α (β, r)

fst :: RLens (α,β) α
fst = Iso $(\lambda(a, b) \to (a, b))$ $(\lambda(a, b) \to (a, b))$

div :: Int $\to$ RLens Int Int
div d = Iso (divMod d) $(\lambda(f, r) \to f * d + r)$

$r = \alpha$	get/set lens
$r = \beta \to \alpha$	store comanad lens
$r = \partial\alpha/\partial\beta$	context lens

Residual lenses

— Definition

$$\textbf{type}\ \text{RLens}\ \alpha\ \beta = \exists r.\ \text{Iso}\ \alpha\ (\beta, r)$$

$$\text{fst} :: \text{RLens}\ (\alpha,\beta)\ \alpha$$
$$\text{fst} = \text{Iso}\ (\lambda(a,b) \rightarrow (a,b))\ (\lambda(a,b) \rightarrow (a,b))$$
$$\text{div} :: \text{Int} \rightarrow \text{RLens Int Int}$$
$$\text{div}\ d = \text{Iso}\ (\text{divMod}\ d)\ (\lambda(f,r) \rightarrow f * d + r)$$

$r = \alpha$	get/set lens
$r = \beta \rightarrow \alpha$	store comanad lens
$r = \partial\alpha/\partial\beta$	context lens

Residual lenses

— Definition

type RLens α β = $\exists r.$ Iso α (β, r)

fst :: RLens (α,β) α
fst = Iso $(\lambda(a, b) \to (a, b))$ $(\lambda(a, b) \to (a, b))$

div :: Int $\to$ RLens Int Int
div d = Iso (divMod d) $(\lambda(f, r) \to f * d + r)$

r = α	get/set lens
r = $\beta \to \alpha$	store comanad lens
r = $\partial\alpha/\partial\beta$	context lens

Residual lenses – Laws

— Definition

$$\textbf{type}\ \text{RLens}\ \alpha\ \beta = \exists r.\ \text{Iso}\ \alpha\ (\beta, r)$$

$$\text{get}\ \ell = \text{fst} \circ \text{apply}\ \ell$$
$$\text{modify}\ \ell\ f = \text{apply}\ (\text{inverse}\ \ell) \circ (\lambda(b, r) \to (f\ \ b, r)) \circ \text{apply}\ \ell$$

— Lens laws

$$\text{modify}\ \ell\ \text{id}$$
$$= \text{apply}\ (\text{inverse}\ \ell) \circ (\lambda(b, r) \to (\text{id}\ \ b, r)) \circ \text{apply}\ \ell$$
$$= \text{apply}\ (\text{inverse}\ \ell) \circ \text{id} \circ \text{apply}\ \ell$$
$$= \text{id}$$

Residual lenses – Laws

— Definition

$\textbf{type}\ \mathsf{RLens}\ \alpha\ \beta = \exists r.\, \mathsf{Iso}\ \alpha\ (\beta, r)$

$\mathsf{get}\ \ell = \mathsf{fst} \circ \mathsf{apply}\ \ell$

$\mathsf{modify}\ \ell\ f = \mathsf{apply}\ (\mathsf{inverse}\ \ell) \circ (\lambda(b, r) \rightarrow (f\ \ b, r)) \circ \mathsf{apply}\ \ell$

— Lens laws

$$
\begin{aligned}
&\mathsf{modify}\ \ell\ \mathsf{id} \\
&\quad = \mathsf{apply}\ (\mathsf{inverse}\ \ell) \circ (\lambda(b, r) \rightarrow (\mathsf{id}\ \ b, r)) \circ \mathsf{apply}\ \ell \\
&\quad = \mathsf{apply}\ (\mathsf{inverse}\ \ell) \circ \mathsf{id} \circ \mathsf{apply}\ \ell \\
&\quad = \mathsf{id}
\end{aligned}
$$

IN·DEI·NOMINE

Residual lenses – Laws

— *Definition*

$$\textbf{type}\ \text{RLens}\ \alpha\ \beta = \exists r.\ \text{Iso}\ \alpha\ (\beta, r)$$

$$\text{get}\ \ell = \text{fst} \circ \text{apply}\ \ell$$
$$\text{modify}\ \ell\ f = \text{apply}\ (\text{inverse}\ \ell) \circ (\lambda(b, r) \to (f\ \ b, r)) \circ \text{apply}\ \ell$$

— *Lens laws*

$$\begin{aligned}
&\text{modify}\ \ell\ f \circ \text{modify}\ \ell\ g \\
&\quad = \text{apply}\ (\text{inverse}\ \ell) \circ (\lambda(b, r) \to (f\ \ b, r)) \circ \text{apply}\ \ell \\
&\qquad\qquad \circ\ \text{apply}\ (\text{inverse}\ \ell) \circ (\lambda(b, r) \to (g\ \ b, r)) \circ \text{apply}\ \ell \\
&\quad = \text{apply}\ (\text{inverse}\ \ell) \circ (\lambda(b, r) \to (f\ \ b, r)) \\
&\qquad\qquad \circ\ (\lambda(b, r) \to (g\ \ b, r)) \circ \text{apply}\ \ell \\
&\quad = \text{apply}\ (\text{inverse}\ \ell) \circ (\lambda(b, r) \to ((f \circ g)\ \ b, r)) \circ \text{apply}\ \ell \\
&\quad = \text{modify}\ \ell\ (f \circ g)
\end{aligned}$$

Residual lenses – Laws

— Definition

$\textbf{type}\ \mathrm{RLens}\ \alpha\ \beta = \exists r.\ \mathrm{Iso}\ \alpha\ (\beta, r)$

$\mathrm{get}\ \ell = \mathrm{fst} \circ \mathrm{apply}\ \ell$

$\mathrm{modify}\ \ell\ f = \mathrm{apply}\ (\mathrm{inverse}\ \ell) \circ (\lambda(b, r) \rightarrow (f\ \ b, r)) \circ \mathrm{apply}\ \ell$

— Lens laws

$$
\begin{aligned}
&\mathrm{get}\ \ell \circ \mathrm{modify}\ \ell\ f \\
&\quad = \mathrm{fst} \circ \mathrm{apply}\ \ell \\
&\qquad\qquad \circ\ \mathrm{apply}\ (\mathrm{inverse}\ \ell) \circ (\lambda(b, r) \rightarrow (f\ \ b, r)) \circ \mathrm{apply}\ \ell \\
&\quad = \mathrm{fst} \circ (\lambda(b, r) \rightarrow (f\ \ b, r)) \circ \mathrm{apply}\ \ell \\
&\quad = f \circ \mathrm{fst} \circ \mathrm{apply}\ \ell \\
&\quad = f \circ \mathrm{get}\ \ell
\end{aligned}
$$

Residual lenses – Laws

— Definition

$\textbf{type}\ \text{RLens}\ \alpha\ \beta = \exists r.\ \text{Iso}\ \alpha\ (\beta, r)$

$\text{get}\ \ell = \text{fst} \circ \text{apply}\ \ell$

$\text{modify}\ \ell\ f = \text{apply}\ (\text{inverse}\ \ell) \circ (\lambda(b, r) \rightarrow (f\ \ b, r)) \circ \text{apply}\ \ell$

— Lens laws

$$
\begin{aligned}
&\text{modify}\ \ell\ (\text{const}\ (\text{get}\ \ell\ a))\ a \\
&\quad = (\text{apply}\ (\text{inverse}\ \ell) \circ (\lambda(b, r) \rightarrow (\text{const}\ (\text{fst}\ (\text{apply}\ \ell\ a))\ b, r)) \\
&\qquad\qquad \circ\ \text{apply}\ \ell)\ a \\
&\quad = \text{apply}\ (\text{inverse}\ \ell)\ ((\lambda(b, r) \rightarrow (\text{fst}\ (\text{apply}\ \ell\ a), r))\ (\text{apply}\ \ell\ a)) \\
&\quad = \text{apply}\ (\text{inverse}\ \ell)\ (\text{apply}\ \ell\ a) \\
&\quad = a
\end{aligned}
$$

Functor transformer lenses (van Laarhoven lenses)

— Definition

type FTLens α β $= \forall f.$ Functor $f \Rightarrow (\beta \to f\,\beta) \to (\alpha \to f\,\alpha)$

class Functor f **where**
  fmap :: $(\alpha \to \beta) \to (\mathrm{f}\ \alpha \to \mathrm{f}\ \beta)$

— Intuition

A lens is a function that turns an embedding of β in functor f into an embedding of α in functor f.

Functor transformer lenses (van Laarhoven lenses)

— Definition

type FTLens α $\beta = \forall f.$ Functor $f \Rightarrow (\beta \to f \beta) \to (\alpha \to f \alpha)$

```
class Functor f where
  fmap :: (α → β) → (f α → f β)
```

— Intuition

A lens is a function that turns an embedding of β in functor f into an embedding of α in functor f.

Functor transformer lenses (van Laarhoven lenses)

— Definition

type FTLens α β = $\forall f.$ Functor $f \Rightarrow (\beta \to f\,\beta) \to (\alpha \to f\,\alpha)$

class Functor f **where**
  fmap :: $(\alpha \to \beta) \to (\text{f}\ \alpha \to \text{f}\ \beta)$

— Implementing (cps) get

type Const γ α = γ
instance Functor (Const γ) **where**
  fmap f x = x

$$\text{FTLens}\ \alpha\ \beta \subseteq (\beta \to \text{Const}\ \gamma\ \beta) \to (\alpha \to \text{Const}\ \gamma\ \alpha)$$
$$= (\beta \to \gamma) \to (\alpha \to \gamma)$$

Functor transformer lenses (van Laarhoven lenses)

— Definition

type FTLens $\alpha\ \beta = \forall f.\, \text{Functor}\ f \Rightarrow (\beta \to f\,\beta) \to (\alpha \to f\,\alpha)$

class Functor f **where**

  fmap :: $(\alpha \to \beta) \to (\text{f}\ \alpha \to \text{f}\ \beta)$

— Implementing modify

type Identity $\alpha = \alpha$

instance Functor (Identity γ) **where**

  fmap $f\ x = f\ x$

$$\begin{aligned}\text{FTLens}\ \alpha\ \beta &\subseteq (\beta \to \text{Identity}\ \beta) \to (\alpha \to \text{Identity}\ \alpha)\\ &= (\beta \to \beta) \to (\alpha \to \alpha)\end{aligned}$$

Functor transformer lenses (cont.)

— Definition

type FTLens α $\beta = \forall f.\, \text{Functor}\ f \Rightarrow (\beta \to f\, \beta) \to (\alpha \to f\, \alpha)$

— (get,set) $\to$ FTLens

fromGetSet :: Functor f $\Rightarrow (\alpha \to \beta) \to (\beta \to \alpha \to \alpha)$
$\to (\beta \to \text{f}\ \beta) \to \alpha \to \text{f}\ \alpha$

fromGetSet get set return a =
fmap (flip set a) (return (get a))

Functor transformer lenses (cont.)

— *Definition*

$$\textbf{type}\ \text{FTLens}\ \alpha\ \beta = \forall f.\ \text{Functor}\ f \Rightarrow (\beta \rightarrow f\,\beta) \rightarrow (\alpha \rightarrow f\,\alpha)$$

— *(get,set) → FTLens*

$$\text{fromGetSet} :: \text{Functor f} \Rightarrow (\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \alpha \rightarrow \alpha) \rightarrow (\beta \rightarrow \text{f}\ \beta) \rightarrow \alpha \rightarrow \text{f}\ \alpha$$

fromGetSet get set return a =

fmap (flip set a) (return (get a))

Functor transformer lenses (cont.)

— Definition

type FTLens α β = $\forall f.\,\text{Functor}\ f \Rightarrow (\beta \to f\,\beta) \to (\alpha \to f\,\alpha)$

— (get,set) $\to$ FTLens

fromGetSet :: Functor f $\Rightarrow (\alpha \to \beta) \to (\beta \to \alpha \to \alpha)$
$\to (\beta \to \text{f}\ \beta) \to \alpha \to \text{f}\ \alpha$

fromGetSet get set return a =
fmap (flip set a) (return (get a))

Plates

Multiple children: biplate

```
data Expr = Var Name | Ap Expr Expr | Lam Name Expr
-- Has 0, 1 or 2 subexpressions
children :: Lens Expr [Expr]

-- Generalizing
class Uniplate α where
    children :: Lens α [α]
class Biplate α β where
    children :: Lens α [β]

deep :: Uniplate α ⇒ (α → α) → (α → α)
deep f = f ∘ modify children (map (deep f))
```

Multiple children: biplate

```
data Expr = Var Name | Ap Expr Expr | Lam Name Expr
-- Has 0, 1 or 2 subexpressions
children :: Lens Expr [Expr]

-- Generalizing
class Uniplate α where
    children :: Lens α [α]
class Biplate α β where
    children :: Lens α [β]

deep :: Uniplate α ⇒ (α → α) → (α → α)
deep f = f ∘ modify children (map (deep f))
```

Multiple children: biplate

```
data Expr = Var Name | Ap Expr Expr | Lam Name Expr
-- Has 0, 1 or 2 subexpressions
children :: Lens Expr [Expr]

-- Generalizing
class Uniplate α where
    children :: Lens α [α]
class Biplate α β where
    children :: Lens α [β]

deep :: Uniplate α ⇒ (α → α) → (α → α)
deep f = f ∘ modify children (map (deep f))
```

Actual type of 'children'

— *Oops*

set children [Var "x",Var "y"] (Lam "x" (Var "z")) = ???

children :: $\alpha \to (\exists n \in \mathbb{N}. (\beta^n, \beta^n \to \alpha))$

— *Isomorphic types*

type MultiStore α β = $\exists n \in \mathbb{N}. (\beta^n, \beta^n \to \alpha)$
data MultiStore′ α β = Zero α | Succ β (MultiStore′ $(\beta \to \alpha)$ β)
type AppStore α β = $\forall f.$ Applicative $f \Rightarrow (\beta \to f\ \beta) \to f\ \alpha$

Actual type of 'children'

— Oops

set children [Var "x",Var "y"] (Lam "x" (Var "z")) = ???

children :: $\alpha \to (\exists n \in \mathbb{N}.\, (\beta^n, \beta^n \to \alpha))$

— Isomorphic types

type MultiStore α β = $\exists n \in \mathbb{N}.\, (\beta^n, \beta^n \to \alpha)$
data MultiStore′ α β = Zero α | Succ β (MultiStore′ $(\beta \to \alpha)$ β)
type AppStore α β = $\forall f.$ Applicative $f \Rightarrow (\beta \to f\ \beta) \to f\ \alpha$

Actual type of 'children'

— Oops

set children [Var "x",Var "y"] (Lam "x" (Var "z")) = ???

children :: $\alpha \to (\exists n \in \mathbb{N}.\, (\beta^n, \beta^n \to \alpha))$

— Isomorphic types

type MultiStore α β $= \exists n \in \mathbb{N}.\, (\beta^n, \beta^n \to \alpha)$
data MultiStore′ α β = Zero α | Succ β (MultiStore′ $(\beta \to \alpha)$ β)
type AppStore α β $= \forall f.$ Applicative $f \Rightarrow (\beta \to f\ \beta) \to f\ \alpha$

AppStore ≈ MultiStore

type MultiStore $\alpha\ \beta = \exists n \in \mathbb{N}.\, (\beta^n, \beta^n \to \alpha)$
type AppStore $\alpha\ \beta = \forall f.$ Applicative $f \Rightarrow (\beta \to f\ \beta) \to f\ \alpha$

class Functor $f \Rightarrow$ Applicative f **where**
  pure :: $\alpha \to f\ \alpha$
  ap :: $f\ (\alpha \to \beta) \to f\ \alpha \to f\ \beta$

How can pure and ap be used to produce an $f\ \alpha$?

st :: AppStore $\alpha\ \beta$
st f = pure a
st f = pure $bToA$ ‘ap‘ $f\ b_1$
st f = pure $bbToA$ ‘ap‘ $f\ b_1$ ‘ap‘ $f\ b_2$
st f = pure $bbbToA$ ‘ap‘ $f\ b_1$ ‘ap‘ $f\ b_2$ ‘ap‘ $f\ b_3$

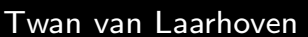
AppStore ≈ MultiStore

type MultiStore α β = $\exists n \in \mathbb{N}. (\beta^n, \beta^n \rightarrow \alpha)$
type AppStore α β = $\forall f.$ Applicative $f \Rightarrow (\beta \rightarrow f\ \beta) \rightarrow f\ \alpha$

class Functor f ⇒ Applicative f **where**
  pure :: $\alpha \rightarrow f\ \alpha$
  ap :: $f\ (\alpha \rightarrow \beta) \rightarrow f\ \alpha \rightarrow f\ \beta$

How can pure and ap be used to produce an $f\ \alpha$?

st :: AppStore α β
st f = pure a
st f = pure $bToA$ ‘ap‘ $f\ b_1$
st f = pure $bbToA$ ‘ap‘ $f\ b_1$ ‘ap‘ $f\ b_2$
st f = pure $bbbToA$ ‘ap‘ $f\ b_1$ ‘ap‘ $f\ b_2$ ‘ap‘ $f\ b_3$



MultiLens

$$\textbf{type}\ \text{MultiLens}\ \alpha\ \beta = \alpha \to \text{AppStore}\ \alpha\ \beta$$
$$= \forall f.\ \text{Applicative}\ f \Rightarrow (\beta \to f\ \beta) \to (\alpha \to f\ \alpha)$$

$$\text{multiGet} :: \text{MultiLens}\ \alpha\ \beta \to \alpha \to [\beta]$$
$$\text{multiModify} :: \text{MultiLens}\ \alpha\ \beta \to (\beta \to \beta) \to \alpha \to \alpha$$

$$\textbf{instance}\ \text{Monoid}\ \gamma \Rightarrow \text{Applicative}\ (\text{Const}\ \gamma)$$

$$\textbf{class}\ \text{Biplate}\ \alpha\ \beta\ \textbf{where}$$
$$\quad \text{children} :: \text{MultiLens}\ \alpha\ \beta$$

Multiplate

— *Multiple 'children'*

$\forall f.\mathrm{Applicative}\ f \Rightarrow (\beta \to f\ \beta, \gamma \to f\ \gamma) \to (\alpha \to f\ \alpha)$

— *Multiple 'parents'*

$\forall f.\mathrm{Applicative}\ f \Rightarrow (\beta \to f\ \beta) \to (\alpha \to f\ \alpha, \delta \to f\ \delta)$

```
data MyPlate f = MyPlate
    { expr :: Expr → f Expr
    , stmt :: Stmt → f Stmt
    , decl :: Decl → f Decl }

multiplate :: ∀f. Applicative f ⇒ MyPlate f → MyPlate f
```

Multiplate

— *Multiple 'children'*

$\forall f.\text{Applicative } f \Rightarrow (\beta \to f\ \beta, \gamma \to f\ \gamma) \to (\alpha \to f\ \alpha)$

— *Multiple 'parents'*

$\forall f.\text{Applicative } f \Rightarrow (\beta \to f\ \beta) \to (\alpha \to f\ \alpha, \delta \to f\ \delta)$

data MyPlate f = MyPlate
  { expr :: Expr → f Expr
  , stmt :: Stmt → f Stmt
  , decl :: Decl → f Decl }

multiplate :: $\forall f.$ Applicative $f \Rightarrow$ MyPlate $f \to$ MyPlate f

The store zoo

Class	# of children
PointedSet	0
Functor	1 (Lens)
PointedFunctor	0 or 1
PreApplicative	1 or more
Applicative	0 or more (Biplate)
Alternative	??
Monad	??

End-of-talk