UNIVERSITY OF CAMBRIDGE

Fun with indexed monads

Dominic Orchard
Computer Laboratory, University of Cambridge

http://dorchard.co.uk/ixmonad

Fun in the Afternoon, March 12th 2014 @ Facebook, London

Monads

Monads

- (design pattern/idiom)

  abstract the composition of computations with *output* effects

- (types)

  distinguish pure from effectful

$$\frac{f : a \to M\ b \qquad g : b \to M\ c}{\lambda x.\ f\ x >>= g\ :\ a \to M\ c}\ \textit{comp.}$$

$$\frac{}{\text{return}\ :\ a \to M\ a}\ \textit{id}$$

Effect systems

$$\frac{\Gamma \vdash e_1 : \tau_1, F \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2, F'}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2, F \cup F'}$$

more fine grained information than monadic typing

Indexed monads

1. cf. “Marriage of monads and effects” (Wadler, Thiemann, 2000)
2. “Coeffects: Unified static analysis of context dependence” (Petricek, Orchard, Mycroft, ICALP 2013) (indexed comonads)
3. “Semantic marriage of monads and effects (extended abstract)” (Orchard, Petricek, Mycroft, arXiV 2013)
4. “Parametric effect monads and semantics of effect systems” (Katsumata, POPL 2014)

Indexed monads

$f : a \to M\ \{\text{read } A\}\ b \qquad g : b \to M\ \{\text{write } B\}\ c$

$g \text{ “o” } f : \ a \to M\ \{\text{read } A, \text{write } B\}\ c$

Monads

$$\frac{f : a \to M\ b \qquad g : b \to M\ c}{\lambda x.\ f\ x >>= g\ :\ a \to M\ c}\ \textit{comp.}$$

$$\frac{}{\mathrm{return}\ :\ a \to M\ a}\ \textit{id}$$

Indexed Monads

$$\frac{f : a \to M\ r\ b \qquad g : b \to M\ s\ c}{\lambda x.\ f\ x >>= g\ :\ a \to M\ (r \bullet s)\ c}\ \textit{comp.}$$

$$\frac{}{\mathsf{return}\ :\ a \to M\ \varepsilon\ a}\ \textit{id}$$

effect indices are a monoid $(F, \bullet, \varepsilon)$

Indexed monads

- (design pattern/idiom)

  abstract the composition of computations with different kinds of *output* effects

- (types)

  distinguish different levels of impurity

```
cabal install ixmonad
```

http://dorchard.co.uk/ixmonad

```
class IxMonad (m :: * → * → *) where
 type Unit m
 type Plus m s t

 return :: a → m (Unit m) a
 (>>=) :: m s a → (a → m t b) → m (Plus m s t) b
```

Indexed reader monad

- Composing ‘reader’ monad is cumbersome
- Indexed reader annotates computation with (named) requirements

$$M\ \{x : A,\ y : B, z : C\}\ a$$

today: $M\ [A, B, C]\ a$

```
foo :: (HCons String HNil) → String

foo = do x ← ask
         return ("Name " ++ x)
```

```
foo :: Show a => HCons String (HCons a HNil) → String

foo = do x  ← ask
         y  ← ask
         return ("Name " ++ x ++ ". Age " ++ (show y))
```

```
*Main> foo (HCons "Dom" (HCons 27 HNil))
"Name Dom. Age 27"
```

```
instance IxMonad (→) where
 type Unit (→)      = HNil
 type Plus (→) s t = Append s t

 return :: a → (HNil → a)
 return x = \HNil → x
```

```
(>>=) :: (s → a) → (a → (t → b)) → (s ++ t → b)
 e >>= k  = \st → let (s, t) = split st
                  in  (k (e s)) t
```

```
split :: (Append s t) → (s, t)
```

Indexed *update* monad

```
foo :: (Put String, Bool)
foo = do put 42
         put "hello"
         return True
```

- Related to the writer monad

$$M\ r\ a = (r, a)$$

Indexed *uupdate** monad

```
foo :: (Put String, Bool)
foo = do put 42
         put "hello"
         return True
```

- Related to the writer monad

$$M\ r\ a = (r, a)$$

* see “Update monads” (Ahmen, Uustalu)

Control.IxMonad.Update

```
data Put a = Put a
data NoPut = NoPut

instance IxMonad (,) where
 type Unit (,)       = NoPut

 type Plus (,) s NoPut    = s
 type Plus (,) s (Put t) = Put t
```

“preserve”

“update”

```
return :: a → (NoPut, a)
return x = (NoPut, x)


(>>=) :: (s, a) → (a → (t, b)) → (Plus (,) s t, b)
e >>= k  = bind e k
```

```
type Plus (,) s NoPut    = s
type Plus (,) s (Put t) = Put t
```

“preserve”

“update”

```
class UpdateBind s t where
  bind :: (s, a) → (a → (t, b)) → (Plus (,) s t, b)

instance UpdateBind s NoPut where
  bind (s, a) k = let (NoPut, b) = k a in (s, b)

instance UpdateBind s (Put t) where
  bind (s, a) k = k a
```

M r may not be a monad

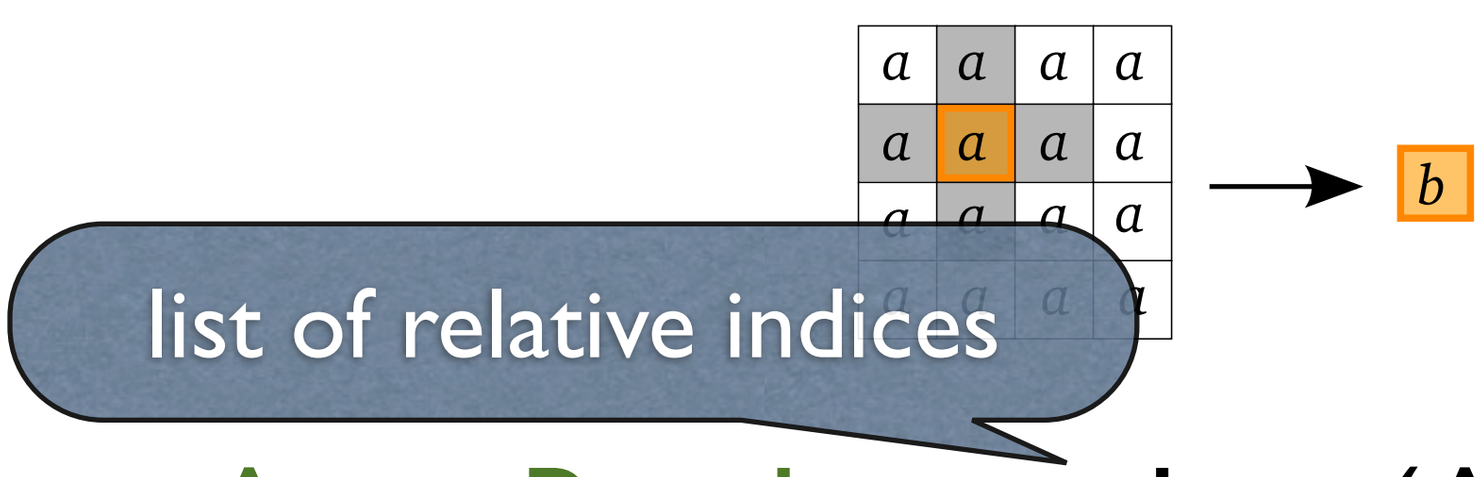
- For the update indexed monad:

$$M\ r\ a = (r, a)$$

- Monad: r must be a monoid
- Indexed monad: must have a monoid on type indices

Effect specifications

- Type signatures to reject code with ‘bad’ effects
- Example: relative array access as effects



ArrayReader a r b = (Array a → b)

http://github.com/dorchard/stencil-specs

ArrayReader a r b = (Array a → b)

```
ix :: TyInt x → ArrayReader a (HCons x HNil) a
```

```
(Pos (S Z)) :: TyInt (S Z)
(Neg (S Z)) :: TyInt (Neg (S Z))
```

```
localMean :: ArrayReader Double
     (HCons Z (HCons (S Z) (HCons (Neg (S Z)) HNil)))
      Double

localMean = do a ← ix (Pos Z)
               b ← ix (Pos (S Z))
               c ← ix (Neg (S Z))
               return $ (a + b + c) / 3.0
```

```
data Stencil r x y where
  Stencil :: ArrayReader x r y → Stencil (Sort r) x y
```

type-level function computing symmetrical relative index list to depth of 1

```
localMean :: Stencil (Symmetrical (S Z)) Double Double

localMean = Stencil $ do a ← ix (Pos Z)
                         b ← ix (Pos (S Z))
                         c ← ix (Neg (S Z))
                         return $ (a + b + c) / 3.0
```

type-level function computing forward relative-index list to depth of 1

```
fwdMean :: Stencil (Forward (S Z)) Double Double

fwdMean = Stencil $ do a ← ix (Pos Z)
                       b ← ix (Pos (S Z))
                       return $ (a + b) / 2.0
```

```
fwdMean' :: Stencil (Forward (S Z)) Double Double

fwdMean' = Stencil $ do a ← ix (Pos Z)
                        b ← ix (Pos (S (S Z)))
                        return $ (a + b) / 2.0
```

```
Couldn't match type `S Z' with `Z'
    Expected type: Stencil (Forward (S Z)) a a
      Actual type: Stencil (Sort (HCons Z (HCons (S (S Z))
HNil))) a a
```

Work in progress

- State {read a} t = a → t
  State {write a} t = (a, t)
  State {read a, write b} t = a → (b, t)
  State {readwrite b} t = b → (b, t)

- Closed-typed families will help
- Implement algebraic effect handlers?

Thanks!

Play: `cabal install ixmonad`

Code: http://github.com/dorchard/ixmonad

http://github.com/dorchard/stencil-specs

Read:

- http://dorchard.co.uk/ixmonad
- "Semantic marriage of monads and effects" (Orchard, Petricek, Mycroft)
- "Parametric effect monads and semantics of effect systems" (Katsumata)

Richer effects

$$\frac{\Gamma \vdash g : \mathbb{B}, \emptyset \qquad \Gamma \vdash e_1 : \tau, F \qquad \Gamma \vdash e_2 : \tau, F'}{\Gamma \vdash \mathbf{if}\ g\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau, F + F'}$$

```
use Control.IxMonad.Cond

see Control.IxMonad.Reader
    Control.IxMonad.Maybe
```