Functional programming with monads combined with comonads

Dominic Orchard

Wednesday 2nd June, 2010

ICFP PC Functional Programming Workshop, MSR, Cambridge

Some functions...

division with possible divide-by-zero exception

$$div : (\mathbb{R}, \mathbb{R}) \to (\mathbb{R} + 1)$$

print a character to stdout

$$putChar : \texttt{Char} \to IO\,()$$

set user state in parser

$$putState : u \to \texttt{ParsecT}\; s\; u\; m\,()$$

Spot the similarity? $f : a \to T\, b$

Monads

T is a monad structure

coproduct (sum) monad (or Maybe) $(\_ + 1)$

$$div : (\mathbb{R}, \mathbb{R}) \rightarrow (\mathbb{R} + 1)$$

IO (state) monad

$$putChar : \texttt{Char} \rightarrow IO\,()$$

Monads

Operations of monad

$$\mu : T(T\,a) \longrightarrow T\,a$$
$$\eta : a \longrightarrow T\,a$$

Kleisli category of a T monad

- Framework for working with morphisms like:

$$f : a \to T\,b$$

called *Kleisli morphisms*

Monads

Given two Kleisli morphisms

$$f : a \to T\, b$$

$$\neq$$

$$g : b \to T\, c$$

Monads

Extension $(\_)^{*} : (a \to T\,b) \to (T\,a \to T\,b)$

$$(\_)^{*} = \mu \circ (T\,f)$$

Lets us compose Kleisli morphisms

$$f : a \to T\,b$$

$$=$$

$$g^{*} : T\,b \to T\,c$$

$$g^{*} \circ f : a \to T\,c$$

Practical Programming with Monads

- Can use compose and/or extension point free e.g.

```
echo = (putChar <.> (const getChar)) ()
```

- What if we want to reuse an intermediate result?

```
echo' = ((\x -> ((\_ -> putChar x)
                   <.> (\_ -> putChar x)) ())
                   <.> (const getChar)) ()
```

Practical Programming with Monads

- *do* notation improves programming with Kleisli morphisms with binding of intermediate results

```
echo = do x <- getChar
          putChar x
          putChar x
```

Practical Programming with Monads

```
do y <- e1    →    extend (\y -> e2) e1
   e2
```

- *extension* happens through `<-`
- binder "*y*" is parameter to Kleisli morphism

Some more functions...

next item in a stream (head of tail)

$$next : Stream\, a \rightarrow a$$

loop body "kernel" function on an array

$$kernel : (Array\, a \times i) \rightarrow a$$

staged computation eval

$$eval : \Box\, a \rightarrow a$$

Spot the similarity? $f : D\, a \rightarrow b$

Comonads

D is a comonad structure

Operations of comonad (dual of a monad)

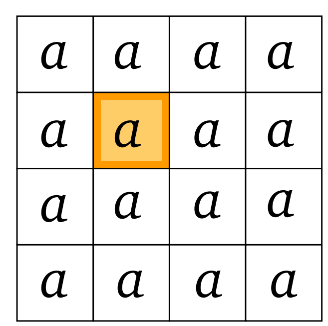
$$\delta : D\, a \longrightarrow D\,(D\, a)$$
$$\epsilon : D\, a \longrightarrow a$$

cf. operations of monad

$$\mu : T(T\, a) \longrightarrow T\, a$$
$$\eta : a \longrightarrow T\, a$$

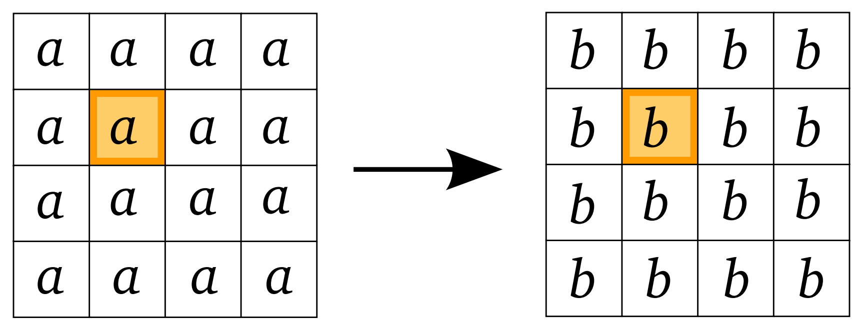
Example comonad: Array

Array is an array with a *cursor*

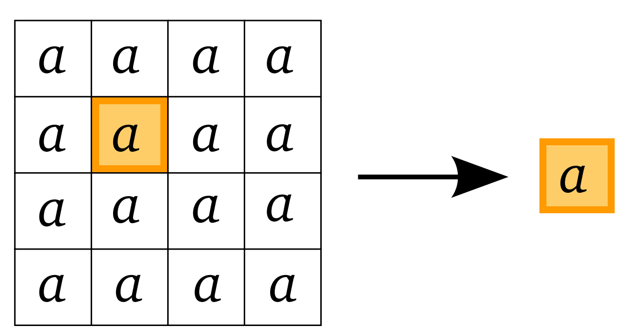


Example comonad: Array

$$fmap : (a \to b) \to \mathbf{Array}\, a \to \mathbf{Array}\, b$$

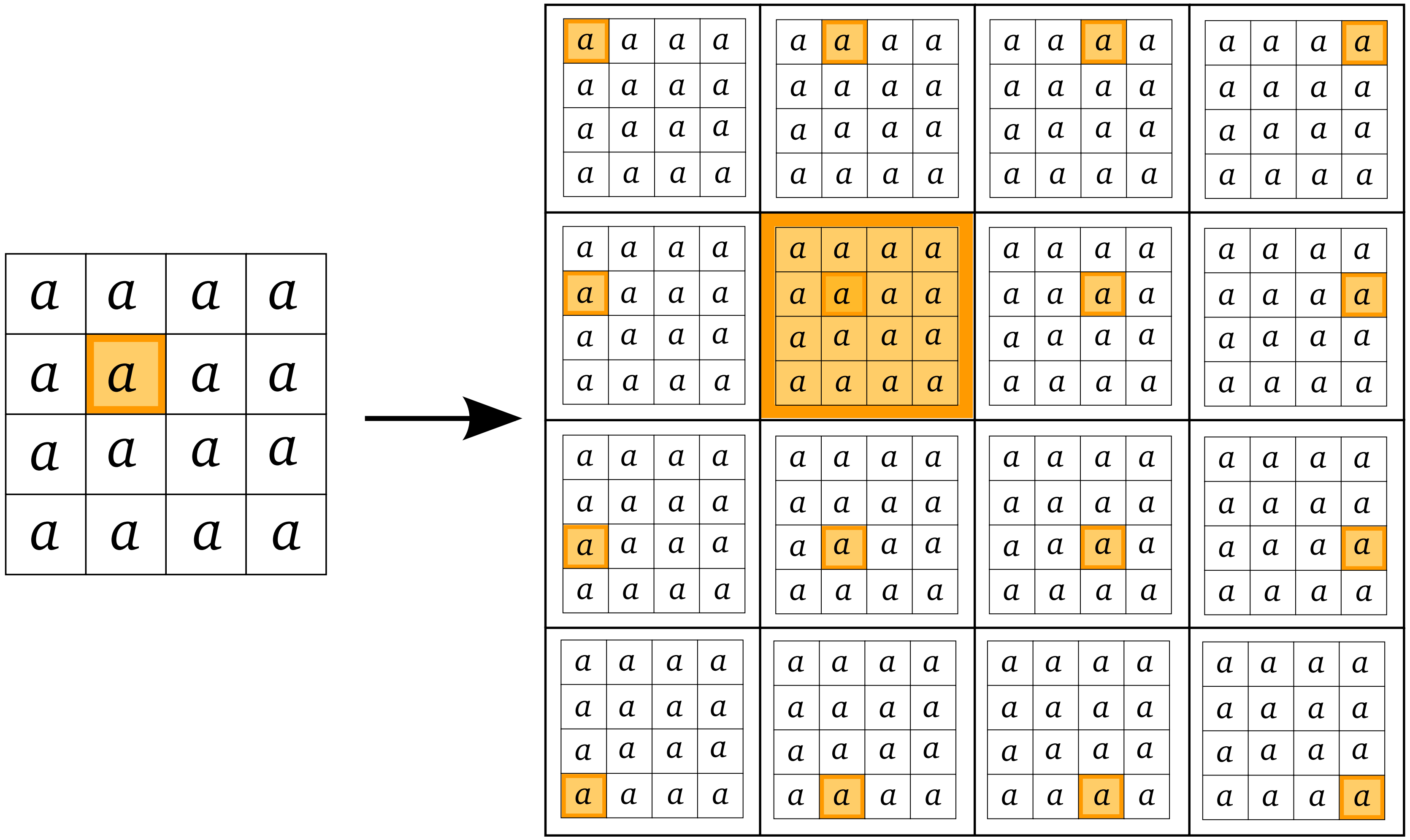


$$\epsilon : \mathbf{Array}\, a \to a$$



Example comonad: Array

$$\delta : \mathbf{Array}\, a \rightarrow \mathbf{Array}(\mathbf{Array}\, a)$$



Comonads

coKleisli category of a D comonad

- Framework for working with morphisms like:

$$f : D\,a \longrightarrow b$$

called co*Kleisli morphisms*

cf. Kleisli morphisms:

$$f : a \rightarrow T\,b$$

Comonads

Extension (coextension)

$$(\_)^{\dagger} : (D\,a \to b) \to (D\,a \to D\,b)$$

$$(\_)^{\dagger} = (D\,f) \circ \delta$$

cf. Kleisli extension

$$(\_)^{*} : (a \to T\,b) \to (T\,a \to T\,b)$$

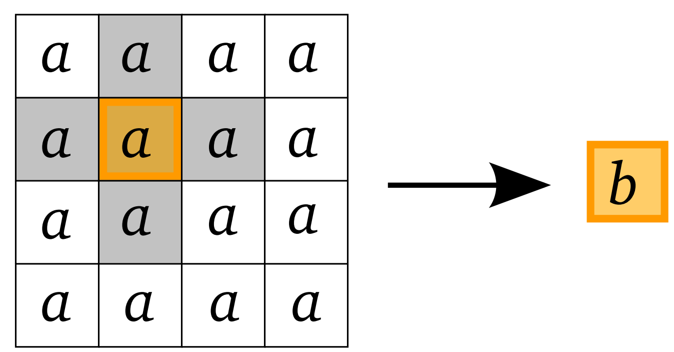
Coextension for CoKleisli composition:

$$f : D\,a \to b \qquad f^{\dagger} : D\,a \to D\,b$$

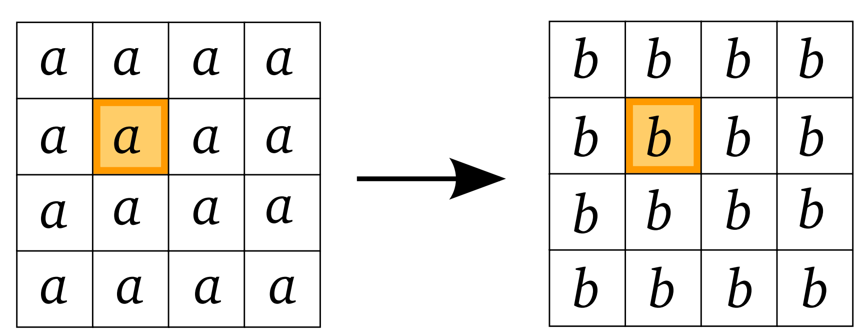
$$g : D\,b \to c \qquad g \circ f^{\dagger} : D\,a \to c$$

Example comonad: Array

$$\textbf{Array}\, a \rightarrow b$$



$$(\_)^{\dagger} : (\textbf{Array}\, a \rightarrow b) \rightarrow (\textbf{Array}\, a \rightarrow \textbf{Array}\, b)$$

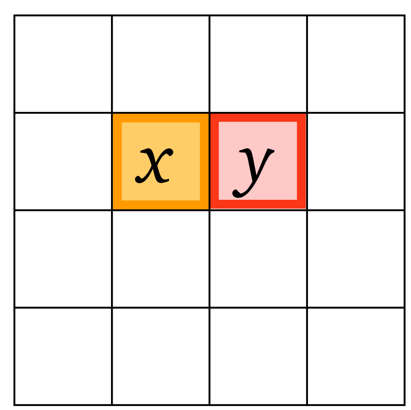


Dr. Jekyll & Mr. Hyde

Recall:

$$div : (\mathbb{R},\ \mathbb{R}) \rightarrow (\mathbb{R} + 1)$$

Consider:

div' 

$= div\,(x, y)$

$$div' : \mathbf{Array}\,\mathbb{R} \rightarrow (\mathbb{R}+1)$$

Coextension of *div'*

$$div' : \mathbf{Array}\,\mathbb{R} \rightarrow (\mathbb{R}+1)$$

Coextension on div':

$$(\_)^{\dagger} : (D\,a \rightarrow b) \rightarrow (D\,a \rightarrow D\,b)$$

$$(div')^{\dagger} : \mathbf{Array}\,\mathbb{R} \rightarrow \mathbf{Array}\,(\mathbb{R}+1)$$

Want to "pull-out" any divide-by-zero exceptions

$$throwExceptions : \mathbf{Array}\,(\mathbb{R}+1) \to ((\mathbf{Array}\,\mathbb{R})+1)$$

Instance of a distributive law of D over T

$$\lambda : D\,T\,a \longrightarrow T\,D\,a$$

BiKleisli Category

BiKleisli category of a T monad and a D comonad

- Framework for working with morphisms like:

$$f : D\,a \rightarrow T\,b$$

called Bi*Kleisli morphisms*

BiKleisli Category

Composition:

$$\circ : (D\, b \to T\, c) \to (D\, a \to T\, b) \to (D\, a \to T\, c)$$

$$g \circ f = (\mathit{extend}\, g) \circ \lambda \circ (\mathit{coextend}\, f)$$

where $\lambda : DT\, a \to TD\, b$

Type check:

$$(\mathit{coextend}\, f) : D\, a \to D(T\, b)$$

$$\lambda \circ (\mathit{coextend}\, f) : D\, a \to T(D\, b)$$

$$(\mathit{extend}\, g) : T(D\, b) \to T\, c$$

$$(\mathit{extend}\, g)\lambda \circ (\mathit{coextend}\, f) : D\, a \to T\, c$$

[Harmer, Hyland, et al. '07, Uustalu & Vene '05, Power & Watanabe '02, Brookes & Stone '93]

But is just BiKleisli composition enough?

$f : \mathbf{Array}\,\mathbb{R} \longrightarrow (\mathbb{R}{+}1) \qquad g : \mathbf{Array}\,\mathbb{R} \longrightarrow (\mathbb{R}{+}1)$

$$g \circ f : \mathbf{Array}\,\mathbb{R} \longrightarrow (\mathbb{R}{+}1)$$

We want:

A comonadic result, not just a single monadic value

Biextend

$$(\_)^{\sharp} : (D\,a \to T\,b) \to T(D\,a) \to T(D\,b)$$

$$f^{\sharp} = \mathit{extend}\,(\lambda \circ \mathit{coextend}\,f)$$

- Derived from a coKleisli category on a Kleisli category
- Perform extension operations through both layers of category, using lambda to get consistent types

Biextend

$$(\_)^\sharp : (D\,a \longrightarrow T\,b) \longrightarrow T(D\,a) \longrightarrow T(D\,b)$$

E.g. biextend on div':

$$div' : \mathbf{Array}\,\mathbb{R} \longrightarrow (\mathbb{R}+1)$$

$$(div')^\sharp : ((\mathbf{Array}\,\mathbb{R})+1) \rightarrow ((\mathbf{Array}\,\mathbb{R})+1)$$

Biextend

$$(\_)^{\sharp} : (D\, a \longrightarrow T\, b) \longrightarrow T(D\, a) \longrightarrow T(D\, b)$$

Can derive composition from biextend

$$g \circ f = (T\epsilon) \circ (biextend\; g) \circ (biextend\; f) \circ \eta_D$$

$$g^{\sharp} \circ f^{\sharp} : T(D\, a) \longrightarrow T(D\, c)$$

$$T\epsilon : T(D\, c) \longrightarrow T\, c$$

$$\eta_D : D\, a \longrightarrow T(D\, a)$$

Biextend'

$$biextend : (D\,a \rightarrow T\,b) \rightarrow T(D\,a) \rightarrow T(D\,b)$$

$$biextend' \;:\; (D\,a \rightarrow T\,b) \rightarrow D(T\,a) \rightarrow D\,(T\,b)$$
$$biextend'\;f = coextend((extend\,f) \circ \lambda)$$

- Not shown further today
- Idea: structure purely local effects, whereas biextend for effects that become global

Example: effectful arrays

- Mutable arrays in Haskell

$$readArray :: (Ix\, i) \Rightarrow \texttt{IOArray}\ i\ e \to i \to \texttt{IO}\, e$$
$$writeArray :: (Ix\, i) \Rightarrow \texttt{IOArray}\, i\ e \to i \to e \to \texttt{IO}\, ()$$

- Look like biKleisli morphisms
- Semantics of effects and arrays conflated

Example: effectful arrays

- Decouple pure, array semantics from state semantics with Array and State

- Effectful array computations as BiKleislis:

$$\mathbf{Array}\, a \rightarrow \mathbf{State}\, b$$

Example: effectful arrays

- Define just lambda

$$\lambda : \mathbf{Array}\,(\mathbf{State}\,a) \rightarrow \mathbf{State}\,(\mathbf{Array}\,a)$$

```
instance Dist Array State where
   dist (Array (b1, b2) arr c) =
     let
       res = mapM (\c' -> counit (Array (b1, b2) arr c')) [b1..b2]
     in
       extend (\vals ->
         unit (Array (buildArray [b1..b2] vals) c (b1, b2)
       ) res
```

Example: effectful arrays

- Thus we get biextend:

$$biextend : (\mathbf{Array}\, a \rightarrow \mathbf{State}\, b) \rightarrow \mathbf{State}\,(\mathbf{Array}\, a) \rightarrow \mathbf{State}\,(\mathbf{Array}\, b)$$

e.g.

```
laplace :: Array Double -> State Double
...
lowpass :: Array Double -> State Double
...

x' :: State (Array Double)
x' = biextend (laplace <.> lowpass) x
```

Example: effectful arrays

- For real IOArray's cannot define:

$$\lambda : \mathbf{Array}\,(\mathbf{State}\,a) \rightarrow \mathbf{State}\,(\mathbf{Array}\,a)$$

- Memory consistency!
- For IOUArray's also for memory consistency AND element restrictions reasons
- But we can define (a restricted) *biextend*

$$biextend : (\mathbf{Array}\,a \rightarrow \mathbf{State}\,a) \rightarrow \mathbf{State}\,(\mathbf{Array}\,a) \rightarrow \mathbf{State}\,(\mathbf{Array}\,a)$$

Practical programming with monads & comonads?

- Can use point-free style here, e.g. for effectful arrays:

```
x' = biextend (laplace <.> lowpass) x
```

- *do* notation for monads/Kleisli
- let-binding for comonads/coKleisli

Practical programming with monads & comonads?

- What if we want to reuse bound intermediate results?
- Recall biextend:

$$(\_)^{\sharp} : (D\,a \rightarrow T\,b) \rightarrow T(D\,a) \rightarrow T(D\,b)$$
$$f^{\sharp} = \mathit{extend}\,(\lambda \circ \mathit{coextend}\,f)$$

- Solution: use *do* with a "half"-biextend

$$(\_)^{\lambda\dagger} : (D\,a \rightarrow T\,b) \rightarrow (D\,a \rightarrow T(D\,b))$$
$$f^{\lambda\dagger} = \lambda \circ \mathit{coextend}\,f$$

Practical programming with monads & comonads?

- "Half"-biextend (operator $>>==$):

$$(\_)^{\lambda\dagger} : (D\, a \rightarrow T\, b) \rightarrow (D\, a \rightarrow T(D\, b))$$

$$f^{\lambda\dagger} = \lambda \circ \mathit{coextend}\, f$$

- *do* notation completes *biextend* by applying extend over (>>==) in the desugaring of *do*

```
do y <- e1    →   extend (\y -> f >>== y) e1
   f >>== y
              →   extend (\y -> (lambda .  coextend f) y) e1
```

Practical programming with monads & comonads?

- E.g.

```
x'' = do elems <- newListArray (0,9)
                   ([1,5,2,3,4,0,13,8,5,7]::[Double])
         x0 <- return $ Array elems
         printArray x0
         x1 <- lowpass >>== x0
         printArray x1
         x2 <- laplace >>== x1
         printArray x2
         x3 <- convolve >>== x2
         printArray x3
```

Conclusions

- Biextend
  - Good for programming with BiKleislis
  - Allows computation on intermediate values
  - Side-step real world restrictions on abstract nonsense

Further Work

- With monads, programming with *extend is* often easier than programming with μ
- *Extend* produces μ
- Axiomatisation for *biextend* that produces λ?
- Another expressive λ-equivalent idiom?

Further Work

- Experiment with *biextend'* further.

$$biextend : (D\,a \rightarrow T\,b) \rightarrow T(D\,a) \rightarrow T(D\,b)$$

$$biextend' : (D\,a \rightarrow T\,b) \rightarrow D(T\,a) \rightarrow D\,(T\,b)$$

- Dual distributive law?

$$\lambda : DT \rightarrow TD$$

$$\lambda' : TD \rightarrow DT$$

Thank you.