Programming with Comonads and Codo Notation

Dominic Orchard, Thursday 2nd June 2011
Institute of Cybernetics, Tallinn University of Technology, Tallinn, Estonia

monads

About 466,000 results (0.09 seconds)

comonads

About 37,800 results (0.21 seconds)

Monads x12 as popular as comonads?!

Monads and Comonads

- Algebraic structures
- Useful in semantics and programming
- Monads structure/abstract "impure" computations
- Monads now ubiquitous in functional programming, esp. Haskell, with *do*-notation
- Comonads less popular.
- Comonads structure/abstract "context-dependent" computations

Talk outline

- Introduction to comonads in a functional programming
- Some examples
- Two language design approaches based on comonads:
  - ***codo*-notation**: embedded comonadic programming.
  - **Ypnos**: language for efficient, parallel, correct, scientific computing built upon *container* comonads

Denotations as morphisms

$$[\![\Gamma : \tau \vdash e : \tau']\!] : [\![\tau]\!] \to [\![\tau']\!]$$

where $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$

$\tau = (\tau_1 \times \ldots \times \tau_n)$

$[\![-]\!] : \mathbf{Exp} \Rightarrow \mathcal{C}$

$\mathcal{C}$ provides composition of denotations $A \to B$

Denotations as Kleisli morphisms

for languages with *impure* features

$$[\![\, \Gamma : \tau \vdash e : \tau' \,]\!] : [\![\tau]\!] \longrightarrow T\,[\![\tau']\!]$$

where $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$

$\tau = (\tau_1 \times \ldots \times \tau_n)$

$[\![-]\!] : \mathbf{Exp} \Rightarrow \mathcal{C}_T$

T is a monad

$\mathcal{C}_T$ provides composition of denotations $A \rightarrow TB$

[Moggi 1989, 1991]

Monads

Given two monadic functions ("Kleisli morphisms"):

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

$$\neq$$

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

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

Lets us compose monadic functions

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

Monads

Given two monadic functions ("Kleisli morphisms"):

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

$$=$$

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

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

Lets us compose monadic functions

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

Monads

Unit

$$\eta : a \to Ta$$

Wrap a pure value in a "trivial" effect

[M1] $\eta^* \circ f = f$ (left identity)

[M2] $f^* \circ \eta = f$ (right identity)

[M3] $(g^* \circ f)^* = g^* \circ f^*$ (associativity)

Monads

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

$$\eta : a \to T\, a$$

In Haskell:

```
class Monad m where
   return :: a -> m a
   (>>=) :: m a -> (a -> m b) -> m b
```

Monads In Haskell

Example: IO monad (encapsulates IO side effects)

```
putChar :: Char -> IO ()
getChar :: IO Char
```

```
echo :: IO ()
echo = getChar >>= (\x -> putChar x)
```

```
echoTwice :: IO ()
echoTwice = getChar >>=
              (\x -> (putChar x) >>=
                (\_ -> (putChar x))
```

Haskell do-notation

- do-notation emulates let-binding (in an impure language)

```
do x <- e1
   e2
```

$\sim$

$$let\ x = e_1\ in\ e_2$$

```
do x <- e1
   y <- e2
   e3
```

$\sim$

$$let\ x = e_1\ in$$
$$let\ y = e_2\ in\ e_3$$

Haskell do-notation

$$[\![\texttt{do x <- e1} \quad \texttt{e2}]\!] \Rightarrow [\![\texttt{e1}]\!] \texttt{ >>= \textbackslash x -> } [\![\texttt{e2}]\!]$$

e.g.

```
echoTwice :: IO ()
echoTwice = getChar >>=
                 (\x -> (putChar x) >>=
                   (\_ -> (putChar x))
```

Haskell do-notation

$$[\![\texttt{do x <- e1} \quad \texttt{e2}]\!] \Rightarrow [\![\texttt{e1}]\!] \texttt{ >>= \textbackslash x -> } [\![\texttt{e2}]\!]$$

e.g.

```
echoTwice :: IO ()
echoTwice = do x <- getChar
               putChar x
               putChar x
```

Haskell do-notation

- do laws, consequence of monad laws [M1-3]

```
1. do { x' <- return x                  do { f x
        f x'                  ≡            }
      }

2. do { x <- m                ≡         do { m
        return x }                         }

3. do { y <- do { x <- m                do { x <- m
                  f x                        y <- f x
                }             ≡              g y
        g y                                }
      }
```

- Monads structure/abstract "impure" computations

$$a \rightarrow Tb$$

$$Da \rightarrow b$$

- Comonads structure/abstract "contextual" computations

Denotations as coKleisli morphisms

for languages with *contextual-dependence*

$$[\![\Gamma : \tau \vdash e : \tau']\!] : D\, [\![\tau]\!] \rightarrow [\![\tau']\!]$$

where $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$

$\tau = (\tau_1 \times \ldots \times \tau_n)$

$[\![-]\!] : \mathbf{Exp} \Rightarrow {}_D\mathcal{C}$

D is a comonad

${}_D\mathcal{C}$ provides composition of denotations $DA \rightarrow B$

Context-dependence

$$[\![\Gamma : \tau \vdash e : \tau']\!] : [\![\tau]\!] \to [\![\tau']\!]$$

Context-dependence

$$[\![\Gamma : \tau \vdash e : \tau']\!] : D\,[\![\tau]\!] \to [\![\tau']\!]$$

- *context* n. **1** the circumstances that form the setting for an event, statement, or idea. **2** the parts that come immediately before and after a word or passage and make its meaning clear.

$$[\![\texttt{today it is raining}]\!] : \mathit{Location} \times \mathit{Time} \to \mathbb{B}$$

$$C := _ ;\, e_2 \mid e_1 ;\, _ \mid _ \oplus e_2 \mid e_1 \oplus _ \mid \ldots$$

Comonads

Given two comonadic functions ("coKleisli morphisms"):

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

$$\neq$$

$$g : D\,b \rightarrow c$$

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

Lets us compose comonadic functions:

$$g \,\hat{\circ}\, f = g \circ f^{\dagger} : D\,a \rightarrow c$$

Comonads

Given two comonadic functions ("coKleisli morphisms"):

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

$$=$$

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

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

Lets us compose comonadic functions:

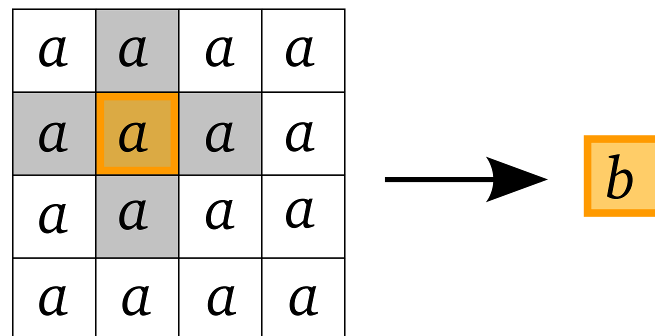
$$g \mathbin{\hat{\circ}} f = g \circ f^{\dagger} : D\,a \to c$$

Example comonad: Array

Array is an array with a *cursor*

a	a	a	a
a	a	a	a
a	a	a	a
a	a	a	a

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



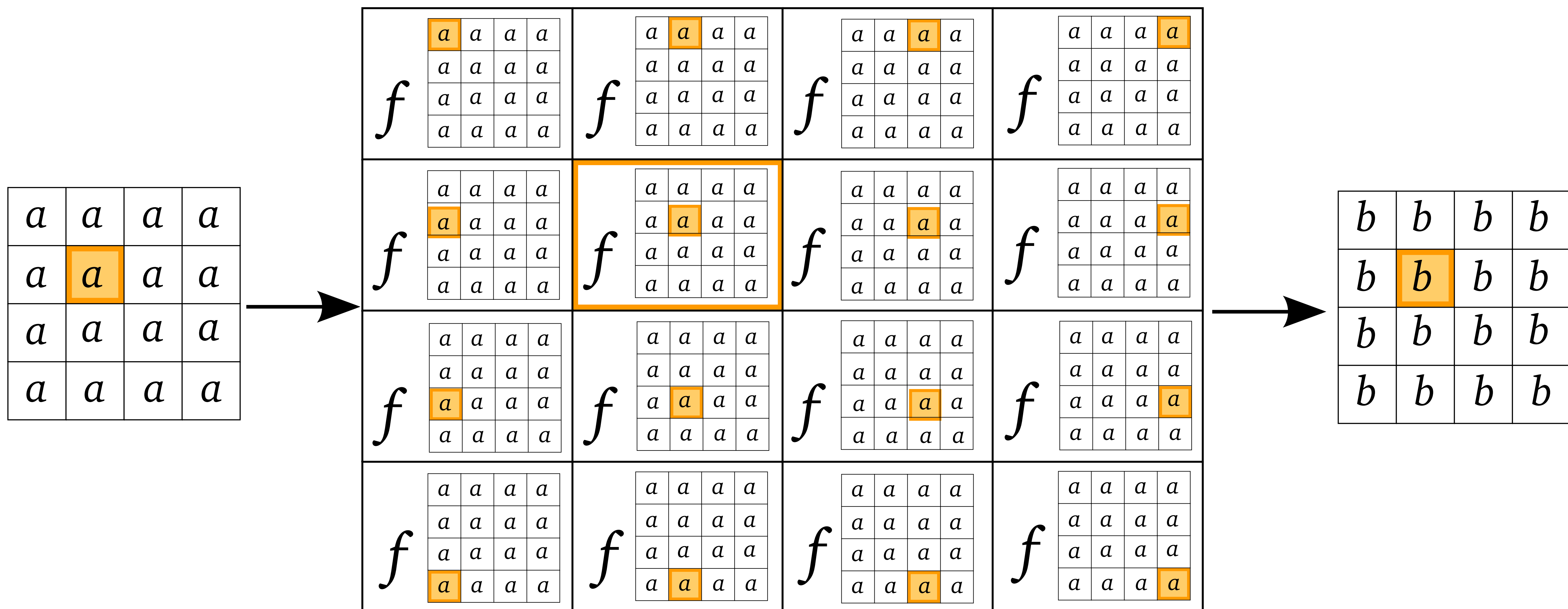
[see "Ypnos: Declarative, Parallel Structured Grid Programming", Orchard, Bolingbroke, Mycroft'10]

Example comonad: Array

- Coextension

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

- Generalised map e.g. convolution on arrays:



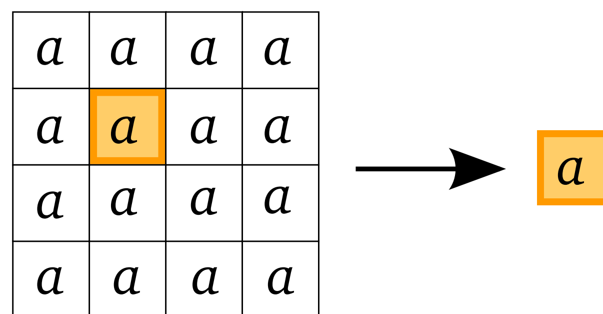
Example comonad: Array

- Counit $\epsilon : Da \rightarrow a$

- Extract the value at the "current context"

  e.g.

$$\epsilon : \textbf{Array}\, a \rightarrow a$$



Comonads

$$g \mathbin{\hat{\circ}} f = g \circ f^{\dagger} \qquad \hat{id} = \epsilon$$

[C1] $f \circ \epsilon^{\dagger} = f$

Right unit (cf. $f \mathbin{\hat{\circ}} \hat{id} = f$)

[C2] $\epsilon \circ f^{\dagger} = f$

Left unit (cf. $\hat{id} \mathbin{\hat{\circ}} f = f$)

[C3] $(g \circ f^{\dagger})^{\dagger} = g^{\dagger} \circ f^{\dagger}$

Associativity (cf. $(h \mathbin{\hat{\circ}} g) \mathbin{\hat{\circ}} f = h \mathbin{\hat{\circ}} (g \mathbin{\hat{\circ}} f)$)

Contexts of a datatype

- Coextension, applies parameter function at every context position:

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

- What are the (valid) context positions for some data type? e.g.

Contexts of a datatype

e.g. D = finite non-empty lists

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

$$f^{\dagger}\ [x_0|x_1|x_2] = [\ f\ [x_0|x_1|x_2]\ |\ f\ [x_1|x_2]\ |\ f\ [x_2]\ ]$$

- ***Current context position***: First (or root) element
- ***Context***: Remaining elements after current

$$f^{\dagger}\ [\boxed{x_0}|x_1|x_2] = [\ f\ [\boxed{x_0}|x_1|x_2]\ |\ f\ [x_0|\boxed{x_1}|x_2]\ |\ f\ [x_0|x_1|\boxed{x_2}]\ ]$$

- ***Current context position***: marked by "pointer"
- ***Context***: all elements before and after current

Contexts of a datatype

$f^{\dagger}\ \boxed{x_0}\,x_1\,x_2 = \left[\ f\ \boxed{x_0}\,x_1\,x_2 \ \middle|\ f\ x_0\,\boxed{x_1}\,x_2 \ \middle|\ f\ x_0\,x_1\,\boxed{x_2}\ \right]$

- Pointer may be:
  - Structural: (see Huet's "Zipper" [1997] and "Comonadic Functional Attribute Evaluation", Uustalu & Vene [2007])
    - E.g: List of preceding elements, current element, and list of future elements) [a] * a * [a]
  - Indexical
    - E.g. address of current element: [a] * Int

Comonads in Haskell

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

Counit $\epsilon : Da \to a$

```
class Comonad c where
    coreturn :: c a -> a
    (=>>) :: c a -> (c a -> b) -> c b
```

```
cobind :: Comonad c => (c a -> b) -> c a -> c b
```

cf. Monads

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

Comonads in Haskell

```
laplace2D :: Array (Int, Int) Double -> Double
laplace2D (Array arr (i, j)) =    arr!(i, j-1) + arr!(i, j+1)
                               +  arr!(i-1, j) + arr!(i+1, j)
                               -  4*(arr!(i, j))
```

```
filter :: Ix i => (a -> Bool) -> a -> Array i a ->a
filter f x (Array arr i) = if (f arr!i) then arr!i
                          else x
```

```
lapThreshold :: Array (Int, Int) Double -> Array (Int, Int) Double
lapThreshold x = x =>> laplace2D
                   =>> (arrFilter (> 0.8) 0.8)
                   =>> (arrFilter (< 0.2) 0.2)
```

Lucid: Context-dependent language

- One view: equational stream language

$$\begin{aligned}
[\![\mathit{1}]\!] &= \langle 1, 1, 1, \ldots \rangle \\
[\![x + y]\!] &= \langle [\![x]\!]_0 + [\![y]\!]_0, [\![x]\!]_1 + [\![y]\!]_1, [\![x]\!]_2 + [\![y]\!]_2, \ldots \rangle \\
[\![next\ x]\!] &= \langle [\![x]\!]_1, [\![x]\!]_2, \ldots \rangle \\
[\![x\, fby\, y]\!] &= \langle [\![x]\!]_0, [\![y]\!]_0, [\![y]\!]_1, \ldots \rangle
\end{aligned}$$

- E.g. $n = 0\ fby\ (n + 1)$

$$\begin{aligned}
[\![n]\!] &= \langle 0, [\![n]\!]_0 + 1, [\![n]\!]_1 + 1, \ldots \rangle \\
&= \langle 0, 1, 2, \ldots \rangle
\end{aligned}$$

Lucid: Context-dependent language

- "The Essence of Dataflow", Uustalu and Vene, 2005
  - Lucid dataflow structured by stream comonads.
  - Presented as interpreter for Lucid AST.
- Here: (shallow) embedding into Haskell:

$$\mathbf{PStream}\,a = \langle a, a, a, \ldots \rangle \times \mathbb{N}$$

$$\epsilon : \mathbf{PStream}\,a \rightarrow a$$

$$\epsilon\,(\langle x_0, x_1, \ldots \rangle, n) = x_n$$

$$(-)^{\dagger} : (\mathbf{PStream}\,a \rightarrow b) \rightarrow (\mathbf{PStream}\,a \rightarrow \mathbf{PStream}\,b)$$

$$f^{\dagger}\,(\langle x_0, x_1, \ldots \rangle, n) = (\langle f(\langle x_0, x_1, \ldots \rangle, 0), f(\langle x_0, x_1, \ldots \rangle, 1), \ldots \rangle, n)$$

Lucid as a Comonad

$$\mathbf{PStream}\, a = \langle a, a, a, \ldots \rangle \times \mathbb{N}$$

$$plus : Num\, a \Rightarrow \mathbf{PStream}\,(a, a) \rightarrow a$$

$$plus(\langle (x_0, y_0), (x_1, y_1), \ldots \rangle, n) = x_n + y_n$$

$$constant : a \rightarrow \mathbf{PStream}\, a$$

$$constant\ x = (\langle x, x, x, \ldots \rangle, 0)$$

[modified from "The Essence of Dataflow", Uustalu & Vene '05]

Lucid as a Comonad

$$\mathbf{PStream}\, a = \langle a, a, a, \ldots \rangle \times \mathbb{N}$$

$$\mathit{next} : \mathbf{PStream}\, a \rightarrow a$$

$$\mathit{next}\, (\langle x_0, x_1, \ldots \rangle, n) = x_{(n+1)}$$

$$\mathit{fby} : \mathbf{PStream}\, a \rightarrow \mathbf{PStream}\, a \rightarrow a$$

$$\begin{aligned} \mathit{fby}\, (\langle x_0, x_1, \ldots \rangle, n)(\langle y_0, y_1, \ldots \rangle, m) = {} & \mathit{if}\ (n = 0 \wedge m = 0) \\ & \mathit{then}\ x_0 \\ & \mathit{else}\ y_{n-1} \end{aligned}$$

[modified from "The Essence of Dataflow", Uustalu & Vene '05]

Lucid as a Comonad

E.g. $n = 0 \, \mathit{fby} \, (n + 1)$

In Haskell:

```
n = (constant 0) =>>
        (\x -> x `fby` (n =>>
              (\y -> (y `plus` (constant 1)))))
```

Fine for a semantic model, but starts to get ugly for programming

A *do*-notation for comonads?

- *do*-notation with monads is just let-binding in a semantics for an impure language [see Moggi '89, '91]
- We introduce **codo-notation** emulating let-binding in semantics of a contextual language

Pure; Single variable contexts

$$let\ x = e_1\ in\ e_2$$

$$[\![e_1]\!] : A \to B$$
$$[\![e_2]\!] : B \to C$$

cf. Unix pipe

$$[\![let\ x = e_1\ in\ e_2]\!] = [\![e_2]\!] \circ [\![e_1]\!]$$
$$: A \to C$$

Pure; Multi-variable contexts

$$let\ x = e_1\ in\ e_2$$

$$[\![e_1]\!] : A \to B$$

$$[\![e_2]\!] : A \times B \to C$$

cf. Unix pipe with branch

$$[\![let\ x = e_1\ in\ e_2]\!] = [\![e_2]\!] \circ \langle id, [\![e_1]\!] \rangle$$

$$: A \to C$$

where $\langle f, g \rangle : A \to (B \times C)$

$$f : A \to B$$

$$g : A \to C$$

Pure; Multi-variable contexts

with meta-language scoping/environments

$$let\ x = e_1\ in\ e_2$$

$[\![e_1]\!] : B$

$[\![e_2]\!] : C$ x is free in e2

$$[\![let\ x = e_1\ in\ e_2]\!] = (\lambda x.[\![e_2]\!])\ [\![e_1]\!]$$

$$: C$$

Comonadic; Single variable contexts

$$let\ x = e_1\ in\ e_2$$

$$[\![e_1]\!] : DA \to B$$

$$[\![e_2]\!] : DB \to C$$

$$[\![let\ x = e_1\ in\ e_2]\!] = [\![e_2]\!] \circ [\![e_1]\!]^{\dagger}$$

$$: DA \to C$$

Comonadic; Multi-variable contexts

$$let\ x = e_1\ in\ e_2$$

$$[\![e_1]\!] : DA \to B$$
$$[\![e_2]\!] : D(A \times B) \to C$$

$$[\![let\ x = e_1\ in\ e_2]\!] = [\![e_2]\!] \circ \langle \epsilon, [\![e_1]\!] \rangle^{\dagger}$$
$$: DA \to C$$

- Can't use meta-language scoping
- Binding depends on parameter structure ($D\,A$)

[Uustalu and Vene "Comonadic Notions of Computation" 2008]

Codo-notation

- Binding depends on parameter structure ($D\,A$)
- Must have explicit incoming parameter (no-scoping)

```
codo(x) y <- e1
        e2
```

```
codo(x) y <- e1
        z <- e2
        e3
```

cf. *do* notation

```
do y <- e1
   e2
```

```
do y <- e1
   z <- e2
   e3
```

Codo-notation

$$\frac{\Gamma, \texttt{x} : D\, a \vdash \texttt{e}_1 : b}{\Gamma \vdash (\texttt{codo(x)}\ \ \texttt{e}_1) : D\, a \to b}$$

$$\frac{\Gamma, x : D\, a \vdash \texttt{e}_1 : b \qquad \Gamma, x : D\, a,\, y : D\, b \vdash \texttt{e}_2 : c}{\Gamma \vdash (\texttt{codo(x)}\ \ \texttt{y <- e}_1\texttt{;}\ \ \texttt{e}_2) : D\, a \to c}$$

Codo-notation

$$[\![let\ x = e_1\ in\ e_2]\!] = [\![e_2]\!] \circ \langle \epsilon, [\![e_1]\!] \rangle^{\dagger} \ : DA \rightarrow C$$

$$\frac{\Gamma, x : D\,a \vdash \mathtt{e}_1 : b \qquad \Gamma, x : D\,a, y : D\,b \vdash \mathtt{e}_2 : c}{\Gamma \vdash (\mathtt{codo(x)\ y\ \text{<-}\ e}_1\mathtt{;\ e}_2) : D\,a \rightarrow c}$$

$[\![$`codo(x) y <- e1; e2`$]\!] =$

$[\![$`e2`$]\!]$ `. (cobind (pair (coreturn, \x ->` $[\![$`e1`$]\!]$`)))`

where

```
cobind :: Comonad d => (d a -> b) -> d a -> d b
pair :: (a -> b, a -> c) -> a -> (b, c)
```

`x:Da` and `y:Db` must be in free-variable context for e2

Codo-notation

$$[\![let\ x = e_1\ in\ e_2]\!] = [\![e_2]\!] \circ \langle \epsilon, [\![e_1]\!] \rangle^{\dagger} \ : DA \rightarrow C$$

$$\frac{\Gamma, x : D\,a \vdash \mathtt{e}_1 : b \qquad \Gamma, x : D\,a,\ y : D\,b \vdash \mathtt{e}_2 : c}{\Gamma \vdash (\mathtt{codo(x)\ y\ \text{<-}\ e_1;\ e_2}) : D\,a \rightarrow c}$$

$[\![$ `codo(x) y <- e1; e2` $]\!] =$

```
(\gamma -> let x = fmap fst gamma
               y = fmap snd gamma
           in e2) .
           (cobind (pair (coreturn, \x -> e1)))
```

`gamma` $: D\,(a \times b)$

(`fmap fst gamma`) $: D\,a$

(`fmap snd gamma`) $: D\,b$

Codo-notation

$[\![$`codo(x) y <- e1; z <- e2; e3`$]\!] =$

```
(\gamma -> let x = fmap (fst . fst) gamma
               y = fmap (snd . fst) gamma
               z = fmap snd gamma
          in e3) .
(cobind (pair (coreturn,
  (\gamma -> let x = fmap fst gamma
                 y = fmap snd gamma
            in e2)))) .
(cobind (pair (coreturn, \x -> e1)))
```

Examples

Lucid:

$$n = 0 \; fby \; (n + 1)$$

Haskell + *codo*:

```
n = cfix (codo(n)  y <- plus n (constant 1)
                   fby (constant 0) y)
```

where

```
cfix :: (Stream a -> a) -> Stream a
cfix f = fix (cobind f)
```

Examples

Lucid:

$$\mathit{fib} = 1 \; \mathit{fby} \; (1 \; \mathit{fby} \, (\mathit{fib} \; + \; \mathit{next} \; \mathit{fib}))$$

Haskell + *codo*:

```
fib = cfix (codo(fib)  fib’ <- next fib
                       fibn2 <- plus fib’ fib
                       fibn1 <- fby (constant 1) fibn2
                       fibn0 <- fby (constant 1) fibn1
                       coreturn fibn0)
```

Examples

Lucid:

$$howfar(x) = if\ (x = 0)\ then\ 0\ else\ 1 + howfar(next\, x)$$

e.g. $howfar\, \langle 0, 1, 2, 0, 5, 3, 7, 1, 0, \ldots \rangle = \langle 0, 2, 1, 0, 4, 3, 2, 1, 0, \ldots \rangle$

Haskell + *codo*:

```
howfar = codo(xs) xs' <- next xs
                  if (coreturn xs)==0 then 0 else 1+(howfar xs')
```

Haskell:

```
howfar [] = []
howfar (x:xs) = if x==0 then 0:(howfar xs)
                else (1+(head (howfar xs))):(howfar xs)
```

Codo Laws

- codo laws, consequence of comonad laws

```
1. codo(x) { y <- f x                          codo(x) { f x
             coreturn y                    ≡             }
           }

2. codo(x) { x' <- coreturn x                  codo(x) { f x
             f x'                          ≡             }
           }

3. codo(x) { z <- codo(x) { y <- f x           codo(x) { y <- f x
                            g y                          z <- g y
                          } x              ≡             h z
             h z                                       }
           }
```

Codo Laws: Example

```
fib = cfix (codo(fib)  fib' <- next fib
                       fibn2 <- plus fib' fib
                       fibn1 <- fby (constant 1) fibn2
                       fibn0 <- fby (constant 1) fibn1
                       coreturn fibn0)
```

||

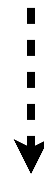
```
fib = cfix (codo(fib)  fibn2 <- (next fib) + (coreturn fib)
                       fibn1 <- fby (constant 1) fibn2
                       fby (constant 1) fibn1)
```

Ypnos

- A language built around *container* comonads
- Aimed at structured/unstructed grid problems in scientific computing
- Highly restricted, good static guarantees on correctness and parallelisability
- Specialised pattern matching syntax abstracts navigation on container

Ypnos: Arrays

```
laplace2D :: Arr (Int, Int) Double -> Double
laplace2D (Arr arr (i, j)) =  arr!(i, j-1) + arr!(i, j+1)
                            +  arr!(i-1, j) + arr!(i+1, j)
                            -  4*(arr!(i, j))
```



```
laplace2D :: Arr (Int, Int) Double -> Double
laplace2D | _  t  _ | = t + l + r + b - 4*c
          | l @c  r |
          | _  b  _ |
```

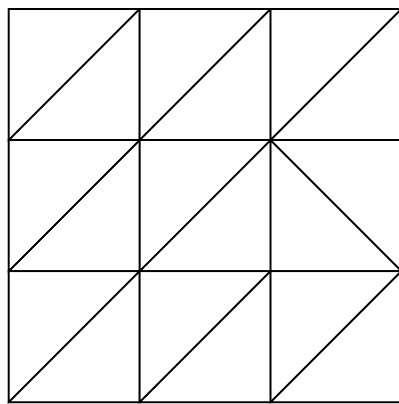
Ypnos: Arrays

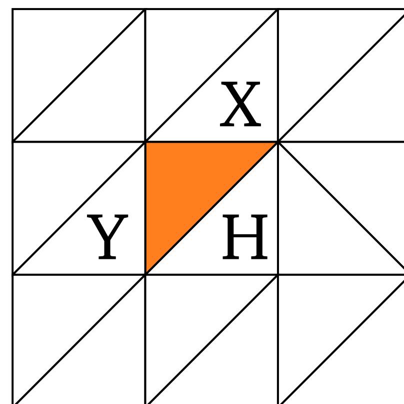
- Grid patterns
  - Static, do not require reduction
  - Provide compile-time information on indexing
  - Check safety properties (out-of-bounds) easily at compiler-time
  - Desugar into relative *navigation operations*:

```
getN :: Array (Int, Int) a -> a
getE :: Array (Int, Int) a -> a
getW :: Array (Int, Int) a -> a
getS :: Array (Int, Int) a -> a
```

Ypnos: Other containers (Meshes)

- Many scientific applications: triangle/polygon meshes for better 2D surface of 3D shapes.






```
getX :: TriangleMesh a -> Maybe a
getY :: TriangleMesh a -> Maybe a
getH :: TriangleMesh a -> Maybe a
```

e.g.

```
meshGaussian :: TriangleMesh Double -> Double
meshGaussian m@(TriangleMesh mesh cursor) =
      ((getX m <|> 0.0) + (getH m <|> 0.0) +
       (getY m <|> 0.0) + 2*(getC m))
```

where `(<|>) :: Maybe a -> a -> a`

```
mesh’ = cobind meshGaussian mesh
```

Ypnos: Other containers (Meshes)

- Hide awkward (error-prone) data types with navigation operations

```
data Orientation = Flip | Same
data TriangleMesh a = TriangleMesh [(a, Maybe Int,
                                        Maybe (Int, Orientation),
                                        Maybe (Int, Orientation))] Int
```

```
meshX = TriangleMesh
          [(0.0, Just 1, Nothing, Just (3, Same)),
           (0.1, Just 0, Just (7, Flip), Just (2, Same)),
           (0.2, Nothing, Just (9, Flip), Just (1, Same)),
           (0.3, Just 4, Nothing, Just (0, Same)),
            ...
```

Ypnos: Other containers (Meshes)

- How to access neighbours of neighbours?

```
getXX :: TriangleMesh a -> Maybe a
getXX mesh = let meshShiftX = cobind getX mesh
             in (join . getX) meshX
```

- Combine with *codo*:

```
meshGaussian2 :: TriangleMesh Double -> Double
meshGaussian2 mesh =
    codo(m) meshX <- getX mesh
            meshY <- getY mesh
            meshH <- getH mesh
            meshXX <- (join . getX) meshX
            meshXH <- (join . getH) meshX
            meshYX <- (join . getX) meshY
            meshYH <- (join . getH) meshY
            meshHX <- (join . getX) meshH
            (!meshXX <|> 0.0 + !meshXH <|> 0.0 + 2*(!meshX <|> 0.0) +
             !meshYX <|> 0.0 + !meshYH <|> 0.0 + 2*(!meshY <|> 0.0) +
             !meshYH <|> 0.0 + !meshHX <|> 0.0 + 4*(!mesh))
```

Conclusion

- Monads and comonads useful in semantics and programming for abstraction
- *do*-notation provides EDSL for impure programming in Haskell, via let-binding with monadic semantics
- New: *codo*-notation provides EDSL for context-dependent programming in Haskell, via let-binding with comonadic semantics
- Conjecture: *codo* even more useful in ML: abstraction for laziness/call-by-name

Conclusion

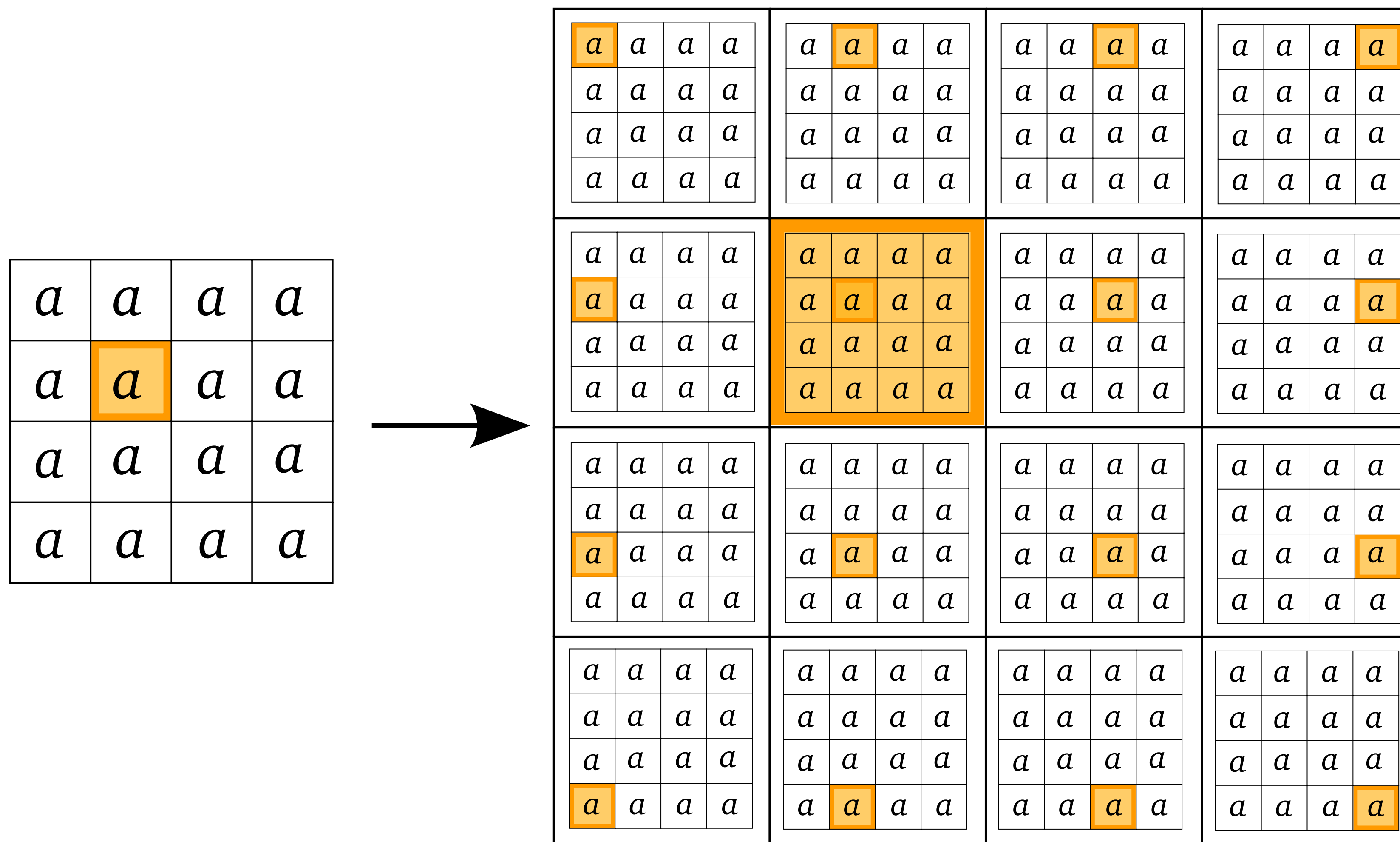
- Ypnos: language for scientific computing
  - Grid patterns used for hiding array access
  - Parameterisable by different container comonads
  - Todo: pattern syntax for non-array structures

Thank you.

Backup Slides

Example comonad: Array

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



Monadic; Single variable contexts

$$let\ x = e_1\ in\ e_2$$

$$[\![e_1]\!] : A \to TB$$
$$[\![e_2]\!] : B \to TC$$

$$[\![let\ x = e_1\ in\ e_2]\!] = [\![e_2]\!]^* \circ [\![e_1]\!]$$
$$: A \to TC$$

Monadic; Multi-variable contexts

$$let\ x = e_1\ in\ e_2$$

$$[\![e_1]\!] : A \to TB$$

$$[\![e_2]\!] : A \times B \to TC$$

$$[\![let\ x = e_1\ in\ e_2]\!] = [\![e_2]\!]^* \circ t \circ \langle id, [\![e_1]\!] \rangle$$

$$: A \to TC$$

$$t : A \times TB \to T(A \times B)$$

Monadic “strength”

[Moggi 1989, 1991]

Monadic; Multi-variable contexts

$$let\ x = e_1\ in\ e_2$$

$$[\![e_1]\!] : TB$$

$$[\![e_2]\!] : TC$$

$$[\![let\ x = e_1\ in\ e_2]\!] = (\lambda x.[\![e_2]\!])^*\ [\![e_1]\!] : TC$$

with meta-language scoping/environments

Do notation:

```
[[do x <- e1
     e2]]      ⇒   [[e1]] >>= \x -> [[e2]]
```

Bido-Notation

Some computations are impure and context-dependent

E.g.

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

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

$$div' \begin{array}{|c|c|c|c|}\hline & & & \\ \hline & x & y & \\ \hline & & & \\ \hline & & & \\ \hline\end{array} = div\,(x, y)$$

Bido-Notation

$$dist : D\,T\,a \rightarrow T\,D\,a$$

$$biextend : (D\,a \rightarrow T\,b) \rightarrow T(D\,a) \rightarrow T(D\,b)$$
$$biextend\ f = bind\,(dist \circ (cobind\ f))$$

$$[\![\texttt{bido(x) y <- e1; e2}]\!] =$$
$$bind\ (\lambda\,\Gamma \rightarrow \mathbf{let}\ x = cmap\ \pi_1\ \Gamma$$
$$y = cmap\ \pi_2\ \Gamma$$
$$\mathbf{in}\ [\![e_1]\!])\ \circ\ dist$$
$$\circ\ \ cobind\,(strength \circ \langle coreturn, \lambda\,x \rightarrow e_1 \rangle)$$