Monadic Effects

Nick Benton

Microsoft Research

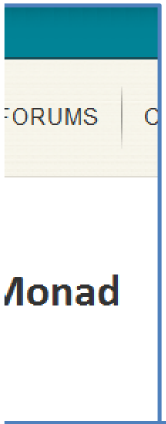
Monad madness

- 1992-08 Monads f or
- 1995 Monadic IO i
- 1999-02 What the a
- 1999 Monads for t e
- 2002 Yet Another las
- 2003-08 All about m
- 2004-07 A Scheme o
- 2004-07 Monads a se
- 2004-08 Monads i
- 2005-07 Monads i
- 2005-11 Of monad
- 2006-03 Understa
- 2006-07 The Mona
- 2006-08 You could
- 2006-09 Meet Bob
- 2006-10 Monad Tr
- 2006-11 There's a
- 2006-12 Maybe M
- 2007-01 Think of a
- 2007-02 Understa
- 2007-02 Crash Cou own in a dark r
- 2007-04 The Real either, but an i
- 2007-03 Monads i utes, using ba
- 2007-07 Monads!
- 2007-08 Monads a entation of m
- 2007-08 Understa
- 2007-08 Monad (s
- 2008-06 Monads (
- 2008 Monads, Cha
- 2009-01 Abstracti
- 2009-03 A Monad Tutorial for Clojure Programmers An interesting perspecti
- 2009-11 What a Monad is not A desperate attempt to end the eternal chain
- 2010-07 I come from Java and want to know what monads are in Haskell - Tim Carstens An example showing how a simple Java class is translated into a
- 2010-08 A Fistful of Monads from Learn You a Haskell An introduction to monads that builds on applicative functors
- 2010-08 Yet Another Monad Tutorial An ongoing sequence of extremely detailed tutorials deriving monads from first principles.

Monads are
- like burritos
- not metaphors
- trees with grafting
- not scary!
- elephants
- promiscuous
- a class of hard drugs
- easy
- monoids
- red herrings
- too much for me

Comp
equiv
progr

Product Details



c
s of LINQ
es not fear

Programming Language Semantics

- Operational
  - $\langle C,S\rangle \Downarrow S'$ or $\langle C,S\rangle \mapsto \langle C',S'\rangle$
  - $(\lambda x.M)\ V \mapsto M[V/x]$
- Denotational
  - Compositional interpretation of syntactic phrases as more abstract mathematical objects
  - What sort of objects affected by
    - syntactic category, or type, of the phrase
    - the language as a whole
    - which aspects of the behaviour of programs we decide to observe
  - Compositionality
    - Denotation has to encode all possible observations arising from placing that phrase in a larger context
    - But want to abstract away from non-observable behaviours; ideally having equal denotations for observationally equivalent things
    - Finding collections of values that have enough information content and structure to interpret phrases, yet do not make too many spurious distinctions, can be hard
    - A good choice embodies a great deal of metatheory about the language before we even consider particular programs

While programs

$$\boxed{c ::= \textsf{skip} \mid x := e \mid c; c \mid \textsf{ifnz}\ e\ \textsf{then}\ c\ \textsf{else}\ c \mid \textsf{while}\ e\ \textsf{do}\ c}$$

Semantics using (partial) functions

$$\begin{aligned}
&\mathit{Store} \overset{def}{=} \mathit{Var} \to \mathbb{Z} \\
&[\![e]\!] : \mathit{Store} \to \mathbb{Z} \\
&[\![c]\!] : \mathit{Store} \rightharpoonup \mathit{Store}
\end{aligned}$$

$$\begin{aligned}
&[\![x := e]\!](s) = s[x \mapsto [\![e]\!](s)] \\
&[\![c_0; c_1]\!] = [\![c_1]\!] \circ [\![c_0]\!] \\
&[\![\textsf{ifnz}\ e\ \textsf{then}\ c_0\ \textsf{else}\ c_1]\!](s) = \begin{cases} [\![c_0]\!](s) & \text{if } [\![e]\!](s) \neq 0 \\ [\![c_1]\!](s) & \text{if } [\![e]\!](s) = 0 \end{cases} \\
&[\![\textsf{while}\ e\ \textsf{do}\ c]\!] = fix\Phi = \textstyle\bigcup_i \Phi^i(\emptyset) \\
&\text{where} \\
&\Phi : (\mathit{Store} \rightharpoonup \mathit{Store}) \to (\mathit{Store} \rightharpoonup \mathit{Store}) \\
&\Phi(f)(s) = \begin{cases} f([\![c]\!](s)) & \text{if } [\![e]\!](s) \neq 0 \\ s & \text{if } [\![e]\!](s) = 0 \end{cases}
\end{aligned}$$

Operational and Denotational:

$$\langle c,\ s\rangle \mapsto^* \langle \mathsf{skip},\ s'\rangle \iff \langle c,\ s\rangle \Downarrow s' \iff [\![c]\!](s) = s'$$

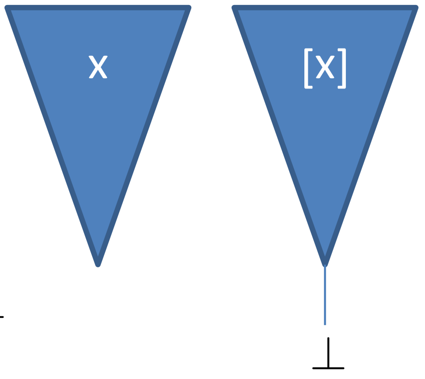
Contextual Equivalence:

$$c \simeq_{ctx} c' \iff \forall C[\cdot]\, s\, s',\ \langle C[c],\ s\rangle \Downarrow s' \iff \langle C[c'],\ s\rangle \Downarrow s'$$

Justifies equations:

$$\begin{array}{l}
(x := 3; y := 5) \simeq_{ctx} (y := 5; x := 3) \\
(\mathsf{ifnz}\ 0\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1) \simeq_{ctx} c_1 \\
((\mathsf{ifnz}\ e\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1); c) \simeq_{ctx} (\mathsf{ifnz}\ e\ \mathsf{then}\ (c_0; c)\ \mathsf{else}\ (c_1; c))
\end{array}$$

Variations



Using ω-cpos instead of sets and partial functions

- Either $[\![c]\!]$ a *strict* (continuous) map $\mathit{Store}_\perp \to \mathit{Store}_\perp$
- Or $[\![c]\!] : \mathit{Store} \to \mathit{Store}_\perp$

In latter case, note $[\![c_0; c_1]\!] = ([\![c_1]\!])^* \circ [\![c_0]\!]$, where if $f : X \to Y_\perp$,

$$f^* : X_\perp \to Y_\perp$$
$$f^*\, a = \begin{cases} f\, x & \text{if } a = [x] \\ \perp & \text{if } a = \perp \end{cases}$$

Adding non-determinism, $\langle c_0 \sqcap c_1,\ s\rangle \mapsto \langle c_0,\ s\rangle$ and $\langle c_0 \sqcap c_1,\ s\rangle \mapsto \langle c_1,\ s\rangle$. Take $[\![c]\!] \in \mathit{Rel}(\mathit{Store}, \mathit{Store})$, i.e. $[\![c]\!] \subseteq (\mathit{Store} \times \mathit{Store})$, with sequential composition interpreted by relational composition

- There's a choice here: $[\![c]\!] = [\![c \sqcap (\textsf{while 1 do skip})]\!]$
- Equivalently, $[\![c]\!] : \mathit{Store} \to \mathbb{P}(\mathit{Store})$, then $[\![c_0; c_1]\!] = ([\![c_1]\!])^* \circ [\![c_0]\!]$ where if $f : X \to \mathbb{P}(Y)$, $f^* : \mathbb{P}(X) \to \mathbb{P}(Y)$ given by $f^*(xs) = \bigcup_{x \in xs} f(x)$

Simple Types

$$A, B := \mathsf{int} \mid \mathsf{unit} \mid A \times B \mid A \to B \mid A + B$$

$$\Gamma, x : A \vdash x : A \qquad \Gamma \vdash \underline{n} : \mathsf{int} \qquad \Gamma \vdash () : \mathsf{unit} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \times B}$$

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathsf{fst}\, M : A} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathsf{snd}\, M : B} \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M N : B}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash (\lambda x : A.M) : A \to B} \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{inl}\, M : A + B} \qquad \frac{\Gamma \vdash N : B}{\Gamma \vdash \mathsf{inr}\, N : A + B}$$

$$\frac{\Gamma \vdash M : A + B \qquad \Gamma, x : A \vdash N : C \qquad \Gamma, y : B \vdash P : C}{\Gamma \vdash \mathsf{case}\ M\ \mathsf{of}\ \mathsf{inl}\, x \Rightarrow N \mid \mathsf{inr}\, y \Rightarrow P : C}$$

$$\frac{E \vdash M : \mathsf{int} \qquad E \vdash M' : \mathsf{int}}{E \vdash M + M' : \mathsf{int}}$$

Operational semantics

Call by value:

$$V := x \mid \lambda x : A.M \mid (V, V) \mid \underline{n} \mid \mathsf{inl}\, V \mid \mathsf{inr}\, V$$

$$\frac{M \Downarrow \lambda x : A.M' \qquad N \Downarrow V \qquad M'[V/x] \Downarrow V'}{M\, N \Downarrow V'} \qquad \frac{M \Downarrow V \qquad N \Downarrow V'}{(M, N) \Downarrow (V, V')}$$

$$\frac{M \Downarrow (V_1, V_2)}{\mathsf{fst}\, M \Downarrow V_1} \qquad \frac{M \Downarrow V}{\mathsf{inl}\, M \Downarrow \mathsf{inl}\, V} \qquad \frac{M \Downarrow \mathsf{inl}\, V \qquad N[V/x] \Downarrow V'}{\mathsf{case}\ M\ \mathsf{of}\ \mathsf{inl}\, x \Rightarrow N \mid \mathsf{inr}\, y \Rightarrow N' \Downarrow V'}$$

$$V \Downarrow V \qquad \frac{M \Downarrow \underline{m} \qquad N \Downarrow \underline{n}}{M + N \Downarrow \underline{m + n}}$$

Call by name:

$$W := \lambda x : A.M \mid (M, M) \mid \underline{n} \mid \mathsf{inl}\, M \mid \mathsf{inr}\, M$$

$$\frac{M \Downarrow \lambda x : A.M' \qquad M'[N/x] \Downarrow W}{M\, N \Downarrow W} \qquad \frac{M \Downarrow (N_1, N_2) \qquad N_1 \Downarrow W}{\mathsf{fst}\, M \Downarrow W}$$

$$\frac{M \Downarrow \mathsf{inl}\, M' \qquad N[M'/x] \Downarrow W}{\mathsf{case}\ M\ \mathsf{of}\ \mathsf{inl}\, x \Rightarrow N \mid \mathsf{inr}\, y \Rightarrow N' \Downarrow W} \qquad \frac{M \Downarrow \underline{m} \qquad N \Downarrow \underline{n}}{M + N \Downarrow \underline{m + n}}$$

Semantics in Set

$$[\![\mathsf{int}]\!] = \mathbb{Z} \qquad [\![\mathsf{unit}]\!] = 1 \qquad [\![A \times B]\!] = [\![A]\!] \times [\![B]\!]$$

$$[\![A \to B]\!] = [\![A]\!] \to [\![B]\!] \ (= [\![B]\!]^{[\![A]\!]}) \qquad [\![A + B]\!] = [\![A]\!] + [\![B]\!]$$

$$[\![x_1 : A_1, \ldots, x_n : A_n]\!] = [\![A_1]\!] \times \cdots [\![A_n]\!]$$

$$[\![\vec{x_i} : \vec{A_i} \vdash x_i : A_i]\!]\rho = \pi_i(\rho) \qquad [\![\Gamma \vdash \underline{n} : \mathsf{int}]\!]\rho = n \qquad [\![\Gamma \vdash () : \mathsf{unit}]\!]\rho = *$$

$$[\![\Gamma \vdash (M, N) : A \times B]\!]\rho = ([\![\Gamma \vdash M : A]\!]\rho,\ [\![\Gamma \vdash N : B]\!]\rho)$$

$$[\![\Gamma \vdash \mathsf{fst}\, M : A]\!]\rho = \pi_1([\![\Gamma \vdash M : A \times B]\!]\rho)$$

$$[\![\Gamma \vdash M\, N : B]\!]\rho = ([\![\Gamma \vdash M : A \to B]\!]\rho)\, ([\![\Gamma \vdash N : A]\!]\rho)$$

$$[\![\Gamma \vdash \lambda x : A.M : A \to B]\!]\rho = \lambda a \in [\![A]\!].([\![\Gamma, x : A \vdash M : B]\!](\rho, a)) \qquad \ldots$$

Equations

$$\Gamma \vdash M \simeq_{ctx} N : A \quad \Longleftrightarrow \quad \forall C[\cdot] : (\Gamma \vdash A) \rhd \mathsf{int}, C[M] \Downarrow \underline{n} \ \Longleftrightarrow \ C[N] \Downarrow \underline{n}$$

beta:

$$(\lambda x : A.M)\, N = M[N/x] \qquad \mathsf{fst}\,(M, N) = M \qquad \mathsf{snd}\,(M, N) = N$$

$$\mathsf{case\ inl}\, M \ \mathsf{of\ inl}\, x \Rightarrow N \mid \mathsf{inr}\, y \Rightarrow N' = N[M/x]$$

$$\mathsf{case\ inr}\, M \ \mathsf{of\ inl}\, x \Rightarrow N \mid \mathsf{inr}\, y \Rightarrow N' = N'[M/y] \qquad M + N = N + M$$

$$\underline{n} + \underline{m} = \underline{n + m} \qquad \ldots$$

eta:

$$M = () \qquad M = \lambda x : A.M\, x \ (a \notin fv(M)) \qquad M = (\mathsf{fst}\, M, \mathsf{snd}\, M)$$

$$\mathsf{case}\ M \ \mathsf{of\ inl}\, x \Rightarrow \mathsf{inl}\, x \mid \mathsf{inr}\, y \Rightarrow \mathsf{inr}\, y = M$$

$$(\text{better: } \mathsf{case}\ M \ \mathsf{of\ inl}\, x \Rightarrow N[\mathsf{inl}\, x/z] \mid \mathsf{inr}\, y \Rightarrow N[\mathsf{inr}\, y/z] = N[M/z])$$

Recursion (hence divergence) in CBV

$$\frac{\Gamma, x : A, f : A \to B \vdash M : B}{\Gamma \vdash (\mathsf{rec}\, f : A \to B\; x = M) : A \to B}$$

$$\frac{M \Downarrow (\mathsf{rec}\, f\; x = M') \qquad N \Downarrow V' \qquad M'[V'/x,\, (\mathsf{rec}\, f\; x = M')/f] \Downarrow V}{M\; N \Downarrow V}$$

$$\text{-} \;= (\mathsf{rec}\, f\; x = f\; x)()$$

$(\lambda x.M)\, N \neq_v M[N/x]$ consider $(\lambda x.())$ - $(\lambda x M)\, V =_v M[V/x]$

$\mathsf{fst}\,(M_1, M_2) \neq_v M_1$ $\mathsf{fst}\,(V_1, V_2) =_v V_1$

$M \neq_v \lambda x.M\, x$ $V =_v \lambda x.V\, x$

Recursion in CBN

$$\frac{\Gamma, x : A \vdash M : A}{\Gamma \vdash (\mathsf{rec} x : A.M) : A} \qquad \frac{M[(\mathsf{rec} x.M)/x] \Downarrow W}{(\mathsf{rec}\, x.M) \Downarrow W}$$

$$\text{-} \;\; = (\mathsf{rec}\, x.x)$$

PCF - observation at ground type$(\lambda x.M)\, N = M[N/x]$

$$\mathsf{fst}\,(M_1, M_2) = M_1$$

$(\lambda x.M\, x) = M$ in particular, $\lambda x.$- $=$ -

$$(\mathsf{fst}\, M, \mathsf{snd}\, M) = M$$

Haskell - observation at all types

$$(\lambda x.M\, x) \neq M$$

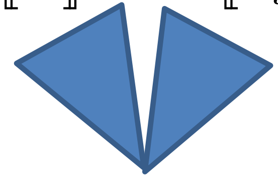
$$(\mathsf{fst}\, M, \mathsf{snd}\, M) \neq M$$

Denotational Semantics CBV

Use pointed ω-cpos and strict maps

$$[\![\mathsf{int}]\!] = \mathbb{Z}_\perp \qquad [\![A \to B]\!] = [\![A]\!] \multimap [\![B]\!] \qquad [\![A \times B]\!] = [\![A]\!] \quad [\![B]\!]$$

$$[\![A + B]\!] = [\![A]\!] \oplus [\![B]\!] \qquad [\![\vec{x_i} : \vec{A_i}]\!] = \bigotimes_i [\![A_i]\!] \qquad [\![\Gamma \vdash M : A]\!] : [\![\Gamma]\!] \to [\![A]\!]$$



Use ω-cpos and explicit lifting

$$[\![\mathsf{int}]\!] = \mathbb{Z} \qquad [\![A \to B]\!] = [\![A]\!] \to ([\![B]\!])_\perp \qquad [\![A \times B]\!] = [\![A]\!] \times [\![B]\!]$$

$$[\![A + B]\!] = [\![A]\!] + [\![B]\!] \qquad [\![\vec{x_i} : \vec{A_i}]\!] = \prod_i [\![A_i]\!] \qquad [\![\Gamma \vdash M : A]\!] : [\![\Gamma]\!] \to ([\![A]\!])_\perp$$

$$[\![\Gamma \vdash \lambda x.M : A \to B]\!] = \Gamma \xrightarrow{cur[\![M]\!]} (A \to B_\perp) \xrightarrow{[\cdot]} (A \to B_\perp)_\perp$$

$$[\![\Gamma \vdash M\,N : B]\!] = \Gamma \xrightarrow{\langle [\![M]\!], [\![N]\!] \rangle} (A \to B_\perp)_\perp \times A_\perp \longrightarrow ((A \to B_\perp) \times A)_\perp \xrightarrow{ev^*} B_\perp$$

Denotational Semantics: CBN

For PCF: Pointed cpos and continuous maps

$$[\![\mathsf{int}]\!] = \mathbb{Z}_\perp \qquad [\![A \to B]\!] = [\![A]\!] \to [\![B]\!] \qquad [\![A \times B]\!] = [\![A]\!] \times [\![B]\!]$$

$$[\![A + B]\!] = ([\![A]\!] + [\![B]\!])_\perp$$

For Haskell: Pointed cpos and continuous maps, more lifting

$$[\![\mathsf{int}]\!] = \mathbb{Z}_\perp \qquad [\![A \to B]\!] = ([\![A]\!] \to [\![B]\!])_\perp \qquad [\![A \times B]\!] = ([\![A]\!] \times [\![B]\!])_\perp$$

$$[\![A + B]\!] = ([\![A]\!] + [\![B]\!])_\perp$$

CBV with global store

$$\Gamma \vdash !X : \mathsf{int} \qquad \qquad \frac{\Gamma \vdash M : \mathsf{int}}{\Gamma \vdash (X := M) : \mathsf{unit}}$$

$$\langle s,\ !X\rangle \Downarrow \langle s,\ \underline{s(X)}\rangle \qquad \qquad \frac{\langle s,\ M\rangle \Downarrow \langle s',\ \underline{n}\rangle}{\langle s,\ X := M\rangle \Downarrow \langle s'[X \mapsto n],\ ()\rangle}$$

$$\frac{\langle s,\ M\rangle \Downarrow \langle s',\ \lambda x.M'\rangle \qquad \langle s',\ N\rangle \Downarrow \langle s'',\ V\rangle \qquad \langle s'',\ M'[V/x]\rangle \Downarrow \langle s''',\ V'\rangle}{\langle s,\ M\,N\rangle \Downarrow \langle s''',\ V'\rangle}$$

Further inequations

$$(\lambda x.\lambda y.(x, y))\, M\, N \neq (\lambda y.\lambda x.(x, y))\, N\, M$$

$$(\lambda x.(x, x))\, M \neq (\lambda x.\lambda y.(x, y))\, M\, M$$

plas various equations involving the new operations.

Denotational

$$[\![\mathsf{int}]\!] = \mathbb{Z} \qquad [\![\mathsf{unit}]\!] = 1 \qquad [\![A \times B]\!] = [\![A]\!] \times [\![B]\!]$$

$$[\![A \to B]\!] = [\![A]\!] \times \mathit{Store} \to [\![B]\!] \times \mathit{Store} \qquad [\![\vec{x_i} : \vec{A_i}]\!] = \prod_i [\![A_i]\!]$$

$$[\![\Gamma \vdash M : A]\!] : [\![\Gamma]\!] \times \mathit{Store} \to [\![A]\!] \times \mathit{Store}$$

$$\begin{aligned}&[\![\Gamma \vdash (M, N) : A \times B]\!]\,(\rho, s) = \\ &\quad ((x, y), s'') \text{ where } [\![N]\!](\rho, s') = (s'', y) \text{ where } [\![M]\!](\rho, s) = (x, s')\end{aligned}$$

$$\Gamma \times S \xrightarrow{\Delta \times 1} \Gamma \times \Gamma \times S \xrightarrow{1 \times [\![M]\!]} \Gamma \times A \times S \xrightarrow{\sigma \times 1} A \times \Gamma \times S \xrightarrow{1 \times [\![N]\!]} A \times B \times S$$

Moggi’s brilliant idea

- The extra structure we add to models of the pure language to deal with these, and many other, notions of side effect always has the same “shape”
- And there are common patterns for just how we use that structure to modify the interpretations of types
- And corresponding patterns apply to the interpretation of terms
- We can capture this commonality by *factoring* our semantics via a new, generic, *computational metalanguage*
- Doing things this way saves repeated work, modularizes, explains, cleans up reasoning by moving side-conditions into the type system, sets us up for further generalizations

The structure

- Separate *values* A from *computations* TA, which may have observable behaviour other than producing a value of type A
- T is *functor* T:C$\rightarrow$ C, so can lift f:A$\rightarrow$ B to Tf:TA$\rightarrow$TB, and this preserves identity and composition
- There's a natural transformation with components η_A:A$\rightarrow$TA which expresses how values may be (uniformly) viewed as trivial computations
- There's a natural transformation μ_A : TTA$\rightarrow$TA that lets us (uniformly) combine effectful behaviours, so we can see a computation of a computation as a computation
- Satisfying some conditions

Monad conditions

$$\begin{tikzcd}
T^3A \arrow[r, "\mu_{TA}"] \arrow[d, "T\mu_A"'] & T^2A \arrow[d, "\mu_A"] \\
T^2A \arrow[r, "\mu_A"'] & TA
\end{tikzcd}
\qquad
\begin{tikzcd}
TA \arrow[r, "T\eta_A"] \arrow[d, "\eta_{TA}"'] \arrow[dr, "id"] & T^2A \arrow[d, "\mu_A"] \\
T^2A \arrow[r, "\mu_A"'] & TA
\end{tikzcd}$$

Strength

$\tau_{A,B} : A \otimes TB \to T(A \otimes B)$

$$\begin{tikzcd}
I \otimes TA \arrow[r, "\tau"] \arrow[dr, "l"'] & T(I \otimes A) \arrow[d, "T(l)"] \\
& TA
\end{tikzcd}
\qquad
\begin{tikzcd}
A \otimes B \arrow[r, "1 \otimes \eta"] \arrow[dr, "\eta"'] & A \otimes TB \arrow[d, "\tau"] \\
& T(A \otimes B)
\end{tikzcd}$$

$$\begin{tikzcd}
(A \otimes B) \otimes TC \arrow[rr, "\tau"] \arrow[d, "\alpha"'] & & T((A \otimes B) \otimes C) \arrow[d, "T(\alpha)"] \\
A \otimes (B \otimes TC) \arrow[r, "1 \otimes \tau"'] & A \otimes T(B \otimes C) \arrow[r, "\tau"'] & T(A \otimes (B \otimes C))
\end{tikzcd}$$

$$\begin{tikzcd}
A \otimes T^2B \arrow[r, "\tau"] \arrow[d, "1 \otimes \mu"'] & T(A \otimes TB) \arrow[r, "T(\tau)"] & T^2(A \otimes B) \arrow[d, "\mu"] \\
A \otimes TB \arrow[rr, "\tau"'] & & T(A \otimes B)
\end{tikzcd}$$

Examples

- Lifting over ω-cpo. $TX = X_\bot$, $\eta(x) = [x]$, $\mu([x]) = x, \mu(\bot) = \bot$
- Nondeterminism. $TX = \mathbb{P}(X)$, $\eta(x) = \{x\}$, $\mu(H) = \bigcup_{S \in H} S$
- Exceptions. $TX = X + E$, $\eta(x) = \mathsf{inl}\,(x)$, $\mu(w) = \mathsf{case}\ w\ \mathsf{of}\ \mathsf{inl}\,w' \Rightarrow w' \mid \mathsf{inr}\,e \Rightarrow \mathsf{inr}\,e$
- State. $TX = S \to X \times S$, $\eta(x) = \lambda s.(x, s)$, $\mu(M) = \lambda s.\, f\, s'$ where $M\, s = (f, s')$
- Read-only state. $TX = S \to X$, $\eta(x) = \lambda s.x$, $\mu(M) = \lambda s.M\, s\, s$
- Output. $TX = X \times M$ for M a monoid. $\eta(x) = (x, \epsilon)$, $\mu((x, m), m') = (x, m \cdot m')$
- Resumptions. $TX = X + TX$, $\eta(x) = \mathsf{inl}\, x$, $\mu(M) = \mathsf{case}\ M\ \mathsf{of}\ \mathsf{inl}\, c \Rightarrow c \mid \mathsf{inr}\, M' \Rightarrow \mathsf{inr}\, \mu(M')$
- Continuations. $TX = (X \to R) \to R$, $\eta(x) = \lambda k.x\, x$, $\mu(M) = \lambda k.\, M\, (\lambda c.c\, k)$

CBV interpretations

$$[\![\mathsf{int}]\!] = \mathbb{Z} \qquad [\![A \to B]\!] = [\![A]\!] \to T([\![B]\!]) \qquad [\![A \times B]\!] = [\![A]\!] \times [\![B]\!]$$

$$[\![A + B]\!] = [\![A]\!] + [\![B]\!] \qquad [\![\vec{x_i} : \vec{A_i}]\!] = \prod_i [\![A_i]\!] \qquad [\![\Gamma \vdash M : A]\!] : [\![\Gamma]\!] \to T([\![A]\!])$$

$$[\![\Gamma \vdash \lambda x.M : A \to B]\!] = \Gamma \xrightarrow{cur[\![M]\!]} (A \to TB) \xrightarrow{\eta} T(A \to TB)$$

$$[\![\Gamma \vdash M\,N : B]\!] =$$

$$\Gamma \xrightarrow{\Delta} \Gamma \times \Gamma \xrightarrow{1 \times [\![M]\!]} \Gamma \times T(A \to TB) \xrightarrow{\tau} T(\Gamma \times (A \to TB))$$

$$. \xrightarrow{T\sigma} T((A \to TB) \times \Gamma) \xrightarrow{T(1 \times [\![N]\!])} T((A \to TB) \times TA)$$

$$. \xrightarrow{T\tau} T^2((A \to TB) \times A) \xrightarrow{T^2 ev} T^3 B \xrightarrow{T\mu} T^2 B \xrightarrow{\mu} TB$$

Kleisli presentation of monads

$T : C \to C \qquad \eta_A : A \to TA \qquad f^* : TA \to TB$ for each $f : A \to TB$

such that $\eta_A^* = 1_{TA}$ and

$$\begin{tikzcd}
A \arrow[r, "\eta_A"] \arrow[dr, "f"'] & TA \arrow[d, "f^*"'] \arrow[dr, "(f;g^*)^*"] & \\
 & TB \arrow[r, "g^*"'] & TC
\end{tikzcd}$$

The formulations are equivalent:

$$\begin{aligned}
(f : A \to TB)^* &= TA \xrightarrow{Tf} T^2B \xrightarrow{\mu_B} TB \\
T(f : A \to B) &= (A \xrightarrow{f} B \xrightarrow{\eta_B} TB)^* \\
\mu_A &= (TA \xrightarrow{1_{TA}} TA)^*
\end{aligned}$$

Parameterized $f : \Gamma \times A \to TB$, $f^* : \Gamma \times TA \to TB$. Precompose with τ.

The computational metalanguage

Extend simple types

$$A ::= \ldots \mid TA$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{val}\, M : TA} \qquad \frac{\Gamma \vdash M : TA \qquad \Gamma, x : A \vdash N : TB}{\Gamma \vdash \mathsf{let}\, x \Leftarrow M \,\mathsf{in}\, N : TB}$$

Interpret in CCC with strong monad/parameterized Kleisli triple

$$[\![\Gamma \vdash \mathsf{val}\, M : TA]\!] = \Gamma \xrightarrow{[\![M]\!]} [\![A]\!] \xrightarrow{\eta} TA$$

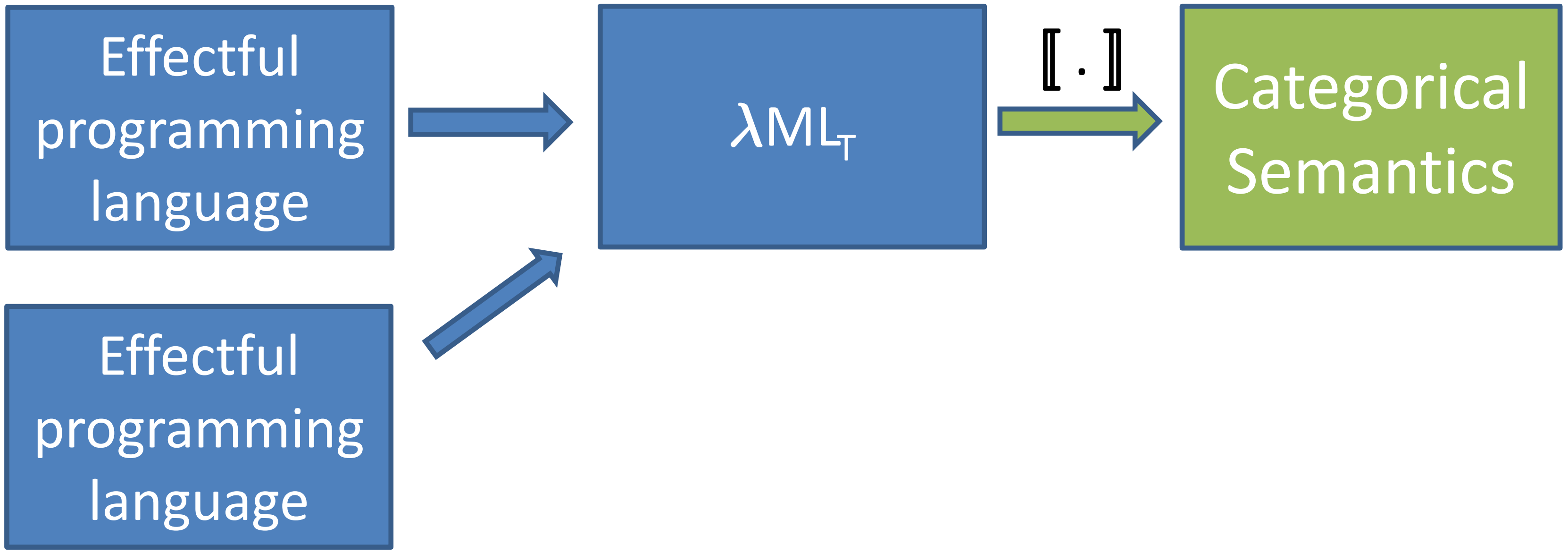
$$[\![\Gamma \vdash \mathsf{let}\, x \Leftarrow M \,\mathsf{in}\, N : TB]\!] = \Gamma \xrightarrow{\Delta} \Gamma \times \Gamma \xrightarrow{1 \times [\![M]\!]} \Gamma \times TA \xrightarrow{[\![N]\!]^*} TB$$

Equations

Full β and η for simple type constructors, plus

$$\mathsf{let}\, x \Leftarrow \mathsf{val}\, M \,\mathsf{in}\, N = N[M/x] \qquad \mathsf{let}\, x \Leftarrow M \,\mathsf{in}\, \mathsf{val}\, x = M$$

$$\mathsf{let}\, x \Leftarrow (\mathsf{let}\, y \Leftarrow M \,\mathsf{in}\, N) \,\mathsf{in}\, P = \mathsf{let}\, y \Leftarrow M \,\mathsf{in}\, \mathsf{let}\, x \Leftarrow N \,\mathsf{in}\, P$$


Effectful programming language
Effectful programming language
λML_T
⟦ · ⟧
Categorical Semantics

CBV translation into λML_T

$$\begin{array}{rcl}
1^\star & = & 1 \\
(X \times Y)^\star & = & X^\star \times Y^\star \\
(X \to Y)^\star & = & X^\star \to TY^\star \\[1ex]
(\Theta \vdash t : X)^\star & = & \Theta^\star \vdash t^\star : TX^\star \\[1ex]
(\Theta, x{:}\, X \vdash x{:}\, X)^\star & = & \Theta^\star, x{:}\, X^\star \vdash [x]{:}\, TX^\star \\
(\Theta \vdash ()\colon 1)^\star & = & \Theta^\star \vdash [()]{:}\, T1 \\
(\Theta \vdash (s,t){:}\, X \times Y)^\star & = & \Theta^\star \vdash \mathrm{let}\ x \leftarrow s^\star\ \mathrm{in}\ \mathrm{let}\ y \leftarrow t^\star\ \mathrm{in}\ [(x,y)]{:}\, T(X^\star \times Y^\star) \\
(\Theta \vdash \mathrm{fst}\ s{:}\, X)^\star & = & \Theta^\star \vdash \mathrm{let}\ z \leftarrow s^\star\ \mathrm{in}\ [\mathrm{fst}\ z]{:}\, TX^\star \\
(\Theta \vdash \mathrm{snd}\ s{:}\, Y)^\star & = & \Theta^\star \vdash \mathrm{let}\ z \leftarrow s^\star\ \mathrm{in}\ [\mathrm{snd}\ z]{:}\, TY^\star \\
(\Theta \vdash \lambda x{:}\, X.\, s{:}\, X \to Y)^\star & = & \Theta^\star \vdash [(\lambda x{:}\, X^\star.\, s^\star)]{:}\, T(X^\star \to TY^\star) \\
(\Theta \vdash s\, t{:}\, Y)^\star & = & \Theta^\star \vdash \mathrm{let}\ z \leftarrow s^\star\ \mathrm{in}\ \mathrm{let}\ x \leftarrow t^\star\ \mathrm{in}\ z\, x{:}\, TY^\star
\end{array}$$

Lifted CBN translation

$$\begin{array}{rcl}
1^\dagger & = & 1 \\
(X \times Y)^\dagger & = & (TX^\dagger \times TY^\dagger) \\
(X \to Y)^\dagger & = & TX^\dagger \to TY^\dagger
\end{array}$$

$$\begin{array}{rcl}
(\Theta \vdash t : X)^\dagger & = & T\Theta^\dagger \vdash t^\dagger : TX^\dagger
\end{array}$$

$$\begin{array}{rcl}
(\Theta, x{:}\, X \vdash x{:}\, X)^\dagger & = & T\Theta^\dagger, x{:}\, TX^\dagger \vdash x{:}\, TX^\dagger \\
(\Theta \vdash ()\,{:}\, 1)^\dagger & = & T\Theta^\dagger \vdash [()]{:}\, T1 \\
(\Theta \vdash (s,t){:}\, X \times Y)^\dagger & = & T\Theta^\dagger \vdash [(s^\dagger, t^\dagger)]{:}\, T(TX^\dagger \times TY^\dagger) \\
(\Theta \vdash \mathrm{fst}\, s{:}\, X)^\dagger & = & T\Theta^\dagger \vdash \mathrm{let}\ z \leftarrow s^\dagger\ \mathrm{in}\ \mathrm{fst}\, z{:}\, TX^\dagger \\
(\Theta \vdash \mathrm{snd}\, s{:}\, Y)^\dagger & = & T\Theta^\dagger \vdash \mathrm{let}\ z \leftarrow s^\dagger\ \mathrm{in}\ \mathrm{snd}\, z{:}\, TY^\dagger \\
(\Theta \vdash \lambda x{:}\, X.\, s{:}\, X \to Y)^\dagger & = & T\Theta^\dagger \vdash [(\lambda x{:}\, TX^\dagger.\, s^\dagger)]{:}\, T(TX^\dagger \to TY^\dagger) \\
(\Theta \vdash s\, t{:}\, Y)^\dagger & = & T\Theta^\dagger \vdash \mathrm{let}\ z \leftarrow s^\dagger\ \mathrm{in}\ z\, t^\dagger{:}\, TY^\dagger
\end{array}$$

CPS translations

Treating CBN and CBV via different translations into common language, rather than via different evaluation orders, already familiar. E.g. for CBV

$$(M\,N)^* = \lambda k.M^*\,(\lambda f.N^*\,(\lambda x.f\,x\,k))$$

With types

$$(A \to B)^* = A^* \to ((B^* \to R) \to R) \quad \cong (B^* \to R) \to (A^* \to R)$$

Operational behaviour of transformed terms matches source, independent of evaluation strategy of target. Full $\beta\eta$ on target proves source equations missed by λ_v.

If we take $TX = (X \to R) \to R$ then monadic translations are just the familiar CPS transformations. Plus get a nicer account of ‘administrative’ reductions.

Kleisli category

Given Kleisli triple $(T, \eta, \cdot^*)$ over C, Kleisli category C_T has

- Objects: same as C
- Morphisms: $C_T(A, B) = C(A, TB)$
- Identities: Identity on A in C_T is $\eta_A : A \to TA$
- Composition: Given $f \in C_T(A, B)$, $g \in C_T(B, C)$, $f; g \in C_T(A, C)$ is $f; g^* : A \to TC$

The conditions on Kleisli triples are just what we need to make this a category. So the CBV interpretation of effectful programs lives in the Kleisli category.

Eilenberg-Moore category

Given monad (T, η, μ) on C, Eilenberg-Moore category C^T has objects T-algebras $\alpha : TA \to A$ st

$$\begin{tikzcd}
T^2A \arrow[r, "\mu_A"] \arrow[d, "T\alpha"'] & TA \arrow[d, "\alpha"] \\
TA \arrow[r, "\alpha"'] & A
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[r, "\eta_A"] \arrow[dr, "\mathrm{id}_A"'] & TA \arrow[d, "\alpha"] \\
 & A
\end{tikzcd}$$

Morphism $(\alpha : TA \to A)$ to $(\beta : TB \to B)$ in C^T is $f : A \to B$ in C st

$$\begin{tikzcd}
TA \arrow[r, "Tf"] \arrow[d, "\alpha"'] & TB \arrow[d, "\beta"] \\
A \arrow[r, "f"'] & B
\end{tikzcd}$$

Algebras

Given single-sorted signature Σ, monad T_Σ on set given by $T_\Sigma(X) =$ the set of Σ terms with variables in X. Then

- $\eta : X \to TX$ includes variables as terms
- A function $f : X \to TY$ is a substitution, assigning a Y-term to each X-variable. The Kleisli lifting $f^* : TX \to TY$ applies the substitution. Can see this as building a term with variables in TY and then flattening.

C^T is just Σ-algebras and homomorphisms. This extends to single-sorted theories

Resolutions

$$\begin{tikzcd}
 & \mathcal{C}^T \arrow[dl, "U"'] \\
\mathcal{C} \arrow[dr, "F"'] & \\
 & \mathcal{C}_T \arrow[uu, hook, "\Phi"']
\end{tikzcd}$$

$U(\alpha : TA \to A) = A$ the carrier of α
$FA = A \qquad Ff = f;\eta$
$\Phi A = \mu_A : T^2 A \to TA$ the free T-algebra on A

$F;\Phi \dashv U$ and $F \dashv \Phi;U$

Both adjunctions induce the original monad T

Relationship with linear logic

- LNL model is symmetric monoidal adjunction between CCC C and SMCC L with F:C$\rightarrow$L left adjoint to G:L$\rightarrow$C
- Comonad ! on L gives model of linear logic, monad on C model of λML_T with commutative monad
- In such a situation the three translations into the metalanguage correspond exactly to three translations into linear logic

Computational Trinitarianism

- Proofs of Propositions (Logic)
- Programs (Terms) of Types (Language)
- Mappings between Structures (Categories)

- So what’s the logical reading of the metalanguage?
  - Take the typing rules and throw away the terms
  - Leaving natural deduction formulation of an intuitionistic modal logic

Natural deduction

$$\frac{}{\Gamma, A \vdash A}\ \mathit{Identity} \qquad \frac{}{\Gamma \vdash \top}\ (\top_{\mathcal{I}})$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B}\ (\supset_{\mathcal{I}}) \qquad \frac{\Gamma \vdash A \supset B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ (\supset_{\mathcal{E}})$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ (\wedge_{\mathcal{I}}) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ (\wedge_{\mathcal{E}}) \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ (\wedge_{\mathcal{E}})$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ (\vee_{\mathcal{I}}) \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ (\vee_{\mathcal{I}}) \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ (\bot_{\mathcal{E}})$$

$$\frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}\ (\vee_{\mathcal{E}})$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash \Diamond A}\ (\Diamond_{\mathcal{I}}) \qquad \frac{\Gamma \vdash \Diamond A \qquad \Gamma, A \vdash \Diamond B}{\Gamma \vdash \Diamond B}\ (\Diamond_{\mathcal{E}})$$

Normalization

- Proof theory of logic forces the equations

$$\dfrac{\dfrac{\begin{matrix}\vdots\\ A\end{matrix}}{\Diamond A}\,(\Diamond_{\mathcal{I}}) \qquad \begin{matrix}[A]\\ \vdots\\ \Diamond B\end{matrix}}{\Diamond B}\,(\Diamond_{\mathcal{E}})$$

which normalises to

$$\begin{matrix}\vdots\\ [A]\\ \vdots\\ \Diamond B.\end{matrix}$$

$$\dfrac{\dfrac{\begin{matrix}\vdots\\ \Diamond A\end{matrix} \qquad \begin{matrix}[A]\\ \vdots\\ \Diamond B\end{matrix}}{\Diamond B}\,(\Diamond_{\mathcal{E}}) \qquad \begin{matrix}[B]\\ \vdots\\ \Diamond C\end{matrix}}{\Diamond C}\,(\Diamond_{\mathcal{E}})$$

commutes to

$$\dfrac{\begin{matrix}\vdots\\ \Diamond A\end{matrix} \qquad \dfrac{\begin{matrix}[A]\\ \vdots\\ \Diamond B\end{matrix} \qquad \begin{matrix}[B]\\ \vdots\\ \Diamond C\end{matrix}}{\Diamond C}\,(\Diamond_{\mathcal{E}})}{\Diamond C}\,(\Diamond_{\mathcal{E}}).$$

Sequent calculus

$$\frac{}{\Gamma, A \vdash A}\ \mathit{Identity} \qquad \frac{\Gamma \vdash B \qquad B, \Gamma \vdash C}{\Gamma \vdash C}\ \mathit{Cut}$$

$$\frac{}{\Gamma, \bot \vdash A}\ (\bot_{\mathcal{L}}) \qquad \frac{}{\Gamma \vdash \top}\ (\top_{\mathcal{R}})$$

$$\frac{\Gamma, A \vdash C}{\Gamma, A \wedge B \vdash C}\ (\wedge_{\mathcal{L}}) \qquad \frac{\Gamma, B \vdash C}{\Gamma, A \wedge B \vdash C}\ (\wedge_{\mathcal{L}}) \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ (\wedge_{\mathcal{R}})$$

$$\frac{\Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma, A \vee B \vdash C}\ (\vee_{\mathcal{L}}) \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ (\vee_{\mathcal{R}}) \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ (\vee_{\mathcal{R}})$$

$$\frac{\Gamma \vdash A \qquad \Gamma, B \vdash C}{\Gamma, A \supset B \vdash C}\ (\supset_{\mathcal{L}}) \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B}\ (\supset_{\mathcal{R}})$$

$$\frac{\Gamma, A \vdash \Diamond B}{\Gamma, \Diamond A \vdash \Diamond B}\ \Diamond_{\mathcal{L}} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash \Diamond A}\ (\Diamond_{\mathcal{R}})$$

Hilbert System

- Usual stuff plus
  - $A \supset \diamond A$
  - $\diamond A \supset ((A \supset \diamond B) \supset \diamond B)$
- Alternatively
  - $A \supset \diamond A$
  - $\diamond\diamond A \supset \diamond A$
  - $(A \supset B) \supset (\diamond A \supset \diamond B)$
- Independently discovered by Fairtlough & Mendler (95), who called this Lax Logic
  - Originally motivated by a range of “true up to constraints” notions in hardware verification

Curry 1952

"The referee has pointed out that for certain kinds of modality it [intro for $\diamond$] is not acceptable … because it allows the proof of

$$\diamond A, \diamond B \vdash \diamond(A \land B).$$

He has proposed a theory of possibility more strictly dual to that of necessity. Although this theory looks promising it will not be developed here."

Models

- CCC plus strong monad, obviously
- But if only interested in proveability, this degenerates to Heyting algebra with a closure operator (inflationary and idempotent)
- Also sound and complete for Kripke models with two relations

$$w \models \Diamond A \text{ iff } \forall v \geq w. \exists u. vRu \text{ and } u \models A.$$

Monad morphisms

Monad morphism $\sigma : (T, \eta, -^*) \to (T', \eta', -^{*'})$ is family $\sigma_A : TA \to T'A$ st

$$\begin{tikzcd}
A \arrow[r, "\eta_A"] \arrow[dr, "\eta'_A"'] & TA \arrow[d, "\sigma_A"] \\
 & T'A
\end{tikzcd}
\qquad
\begin{tikzcd}
TA \arrow[r, "f^*"] \arrow[d, "\sigma_A"'] & TB \arrow[d, "\sigma_B"] \\
T'A \arrow[r, "(f;\sigma_B)^{*'}"'] & T'B
\end{tikzcd}$$

for $f : A \to TB$

(In bijection with carrier preserving functors $V : C^{T'} \to C^T$.)

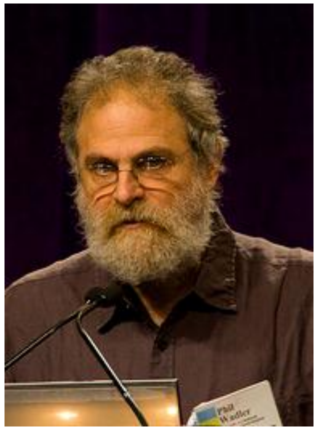
Monad transformers

- Function F mapping monads to monads
- With a monad morphism in_T:T$\rightarrow$ FT for each monad T
- Think of F as adding a new effect to yield T'
- New monad will come with its own operations
- Old operations, general form
  - op: $\forall$ X.A$\rightarrow$ (B$\rightarrow$ TX)$\rightarrow$ TX
- must be lifted to the new monad
  - op': $\forall$ X. A$\rightarrow$ (B$\rightarrow$ T'X)$\rightarrow$ T'X

Structure on the Kleisli category

- Has coproducts if C does (F left adjoint)
- Premonoidal structure functorial in each arg
- Monoidal iff monad is commutative
- Morphisms F(f) commute with anything, they're central
- Premon cat has distinguish SM centre M and id on objects J into premon K, pres prod strcuture
- When M cartesian call it Frey cat

Wadler’s brilliant idea



- Functional programmers had been writing messy programs for a decade or so, doing explicitly what imperative programmers did implicitly
  - Passing around name supplies
  - Passing around states
  - Propagating errors
- Had already come up with list comprehensions along the lines of set comprehensions
- Then saw Moggi’s work and realized that there was a new abstraction that could be used to refactor all these kinds of programs
- And we could pretty much express it in the languages we already had
- Comprehending Monads LFP’90
- The Essence of Functional Programming POPL’92

Monads in Haskell

In Kleisli triple style, take `T : * -> *` to be a Haskell type constructor

```
return :: a -> T a
(>>=) :: T a -> (a -> T b) -> T b
```

So let $x \Leftarrow e_1$ in e_2 becomes

```
e1 >>= \x -> e2
```

For example

```
data Maybe a = Just a | Nothing

return a = Just a

m >>= f = case m of
                   Just a -> f a
                   Nothing -> Nothing

failure = Nothing
```

Failure *is* an option – using the Maybe monad

```
divide :: Maybe Int -> Maybe Int -> Maybe Int
divide a b = a >>= \m ->
             b >>= \n ->
             if n==0 then failure
             else return (a ‘div‘ b)
```

State

Three possibilities

```
type State s a = s -> (s,a)               -- type synonym
newtype State s a = State (s -> (s,a)) -- nominal, unlifted
data State s a = State (s -> (s,a))    -- lazy constructor, lifted

return a = State (\s -> (s,a))
State m >>= f = State (\s -> let (s',a) = m s
                             State m' = f a
                             in m' s')


readState :: State s s
readState = State (\s -> (s,s))

writeState :: s -> State s ()
writeState s = State (\_ -> (s,()))

increment :: State Int ()
increment = readState >>= \s ->
            writeState (s+1)
```

Type classes

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

instance Monad Maybe where
  return a = Just a
  m >>= f = case m of
              Just a -> f a
              Nothing -> Nothing

instance Monad (State s) where
  return a = State (\s -> (s,a))
  State m >>= f = State (\s -> let (s',a) = m s
                                   State m' = f a
                               in m' s')

addM a b = a >>= \m ->
           b >>= \n ->
           return (m+n)
addM :: (Monad m) => m Int -> m Int -> m Int
```

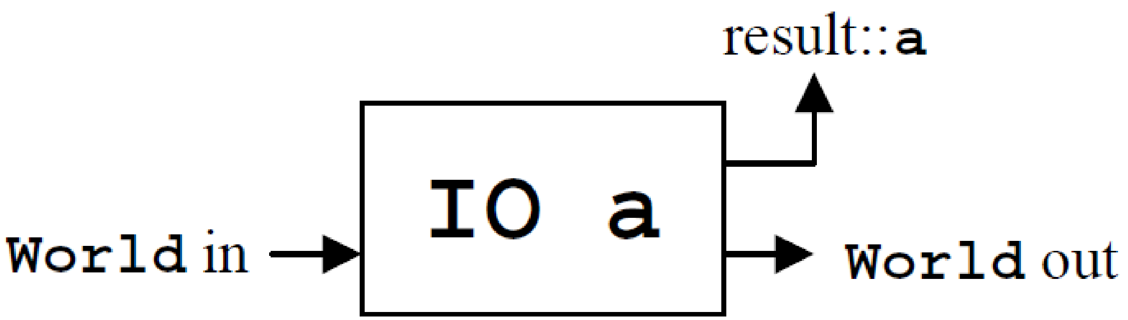
Working with monads

```
liftM :: Monad m => (a -> b) -> m a -> m b
liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c
sequence :: Monad m => [m a] -> m [a]

addM = liftM2 (+)

addM a b = do m <- a
              n <- b
              return (m+n)

do e       =  e

do x <- e  =  e >>= (\x -> do c)
   c

do e       =  e >>= (\_ -> do c)
   c
```

```
data Tree a = Leaf a | Bin (Tree a) (Tree a) deriving Show

unique :: Tree a -> Tree (a,Int)

unique' :: Tree a -> State Int (Tree (a,Int))

tick :: State Int Int
tick = do n <- readState
          writeState (n+1)
          return n

unique' (Leaf a) = do n <- tick
                      return (Leaf (a,n))
unique' (Bin t1 t2) = liftM2 Bin (unique' t1) (unique' t2)

unique t = runState 1 (unique' t)

runState s (State f) = snd (f s)

test3 = unique (Bin (Bin (Leaf 'a') (Leaf 'b')) (Leaf 'c'))

>Bin (Bin (Leaf ('a',1)) (Leaf ('b',2))) (Leaf ('c',3))
```

Peyton Jones and Wadler's brilliant idea

- Lazy functional programmers had been struggling for ages with I/O
- Fundamentally impure – depends on and modifies the state of the world – so breaks all your lovely reasoning principles
- Can't just stick it in and hope for the best like the CBV guys did – evaluation order seriously unpredictable
  - Call by need predicated on the assumption that multiple evaluations always return the same result
- Stream IO, Continuation-based IO, linear types
- Imperative functional programming POPL'93
- We know how to *model* I/O within the language – basically its State Universe
- But within the language we could duplicate, roll back, discard the universe
- BUT if we make the monad abstract and only provide primitives that treat the universe linearly
  - It looks like a functional program to the programmer
  - But can mutate the universe "in place" under the hood
- The IO monad



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

```
data IORef a  -- An abstract type
newIORef    :: a -> IO (IORef a)
readIORef   :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

```
openFile :: String -> IOMode -> IO Handle
hPutStr  :: Handle -> [Char] -> IO ()
hGetLine :: Handle -> IO [Char]
hClose   :: Handle -> IO ()
```

ST monad

- Purely functional code can be asymptotically less efficient than "equivalent" imperative code
- Can use IORefs, but then no way out
- Sometimes want to encapsulate imperative computation within a term that will behave purely functionally
- ST a is like State -> (State,a) except
  - State can hold dynamically allocated typed references
  - It's abstract and can be implemented destructively
  - Its uses can be encapsulated

runST

```
newSTRef :: a -> ST s (STRef s a)
readSTRef :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()
```

`s` is a dummy type variable,or *region*, that can be used to tag references and effects living in different `State`s

```
runST :: (forall s. ST s a) -> a
```

This *rank-2* polymorphic type is the thing that lets us get *out* of the monad. We can only apply it to computations that are parametric in their region, so they cannot import references from the outside or leak them through their result value

Examples

This is OK

```
impure = do x <- newSTRef 0
            y <- readSTRef x
            writeSTRef x (y+1)
            z <- readSTRef x
            return z

test4 = runST impure
```

But these are not

```
runST (newSTRef 0)

runST (do r<-newSTRef 0
               return (runST (readSTRef r)))
```

Monad transformers

- Often want to combine monads, which we do by layering them on top of each other
- Instead of individual monads, work with monad *transformers* that extend an existing monad with a new effect
- Will be of kind `(*->*)->(*->*)`
- Use type class trickery to try to infer as much as possible

MaybeT

```
newtype MaybeT m a = MaybeT (m (Maybe a))

instance Monad m => Monad (MaybeT m) where
 return x = MaybeT (return (Just x))
 MaybeT mm >>= f =
  MaybeT (do x <- mm              -- desugars into m’s >>=
             case x of
               Nothing -> return Nothing
               Just a -> let MaybeT m’ = f a in m’)
```

A class for monad transformers

```haskell
class (Monad m, Monad (t m)) => MonadTransformer t m where
  lift :: m a -> t m a

instance Monad m => MonadTransformer MaybeT m where
  lift m = MaybeT (do x <- m
                      return (Just x))
```

Now need to add operations. The following isn't good enough:

```haskell
failure :: MaybeT m a
handle :: MaybeT m a -> MaybeT m a -> MaybeT m a
```

✗

Maybe-like monads

```
class Monad m => MaybeMonad m where
  failure :: m a
  handle :: m a -> m a -> m a
```

Now anything we get by applying the MaybeT transformer is a MaybeMonad, but later there'll be others too

```
instance Monad m => MaybeMonad (MaybeT m) where
  failure = MaybeT (return Nothing)
  MaybeT m `handle` MaybeT m' =
    MaybeT (do x <- m
               case x of
                 Nothing -> m'
                 Just a -> return (Just a))
```

Recipe

- We define a type to represent the transformer, say `TransT`, with two parameters, the first of which should be a monad.
- We declare `TransT m` to be a `Monad`, under the assumption that `m` already is.
- We declare `TransT` to be an instance of class `MonadTransformer`, thus defining how computations are lifted from `m` to `TransT m`.
- We define a class `TransMonad` of '`Trans`-like monads', containing the operations that `TransT` provides.
- We declare `TransT m` to be an instance of `TransMonad`, thus implementing these operations..

Examples

```
newtype StateT s m a = StateT (s -> m (s, a))
```

```
class Monad m => StateMonad s m | m -> s where
  readState :: m s
  writeState :: s -> m ()
```

```
newtype ContT ans m a = ContT ((a -> m ans) -> m ans)
```

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

Building it up

```
newtype Id a = Id a
```

```
instance MaybeMonad m => MaybeMonad (StateT s m) where
  failure = lift failure
  StateT m ‘handle‘ StateT m’ = StateT (\s -> m s ‘handle‘ m’ s)
```

```
type Parser a = StateT String (MaybeT Id) a
```