Monads (1)

“Shall I be pure or impure?” (Wadler, 1992)
Monads (1)

“Shall I be pure or impure?” (Wadler, 1992)

- Absence of effects
  - makes programs easier to understand and reason about
  - make lazy evaluation viable
  - enhances modularity and reuse.
Monads (1)

“Shall I be pure or impure?” (Wadler, 1992)

- Absence of effects
  - makes programs easier to understand and reason about
  - make lazy evaluation viable
  - enhances modularity and reuse.

- Effects (state, exceptions, . . . ) can
  - yield concise programs
  - facilitate modifications
  - improve the efficiency.
Monads (2)

- Monads bridges the gap: allow effectful programming in a pure setting.
Monads (2)

- Monads bridges the gap: allow effectful programming in a pure setting.
- *Thus we shall be both pure and impure, whatever takes our fancy!*
Monads (2)

- Monads bridges the gap: allow effectful programming in a pure setting.
- *Thus we shall be both pure and impure, whatever takes our fancy!*
- Monads originated in Category Theory.
Monads (2)

- Monads bridges the gap: allow effectful programming in a pure setting.
- *Thus we shall be both pure and impure, whatever takes our fancy!*
- Monads originated in Category Theory.
- Adapted by Moggi for structuring denotational semantics.
Monads (2)

- Monads bridges the gap: allow effectful programming in a pure setting.
- *Thus we shall be both pure and impure, whatever takes our fancy!*
- Monads originated in Category Theory.
- Adapted by Moggi for structuring denotational semantics.
- Adapted by Wadler for structuring functional programs.
Monads (3)

- Key idea of monads: computations as \textit{first-class entities}.
Monads (3)

- Key idea of monads: computations as \textit{first-class entities}.
- Monads promotes disciplined, modular use of effects since the type of a program reflects which effects that occurs.
Monads (3)

- Key idea of monads: computations as *first-class entities*.
- Monads promotes disciplined, modular use of effects since the type of a program reflects which effects that occurs.
- Monads allows us great flexibility in tailoring the effect structure to our precise needs.
First Two Lectures

- Effectful computations: motivating examples
- Monads
- The Haskell `do`-notation
- Some standard monads
- A concurrency monad
Example: A Simple Evaluator

data Exp = Lit Integer
  | Add Exp Exp
  | Sub Exp Exp
  | Mul Exp Exp
  | Div Exp Exp

eval :: Exp -> Integer
eval (Lit n) = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Sub e1 e2) = eval e1 - eval e2
eval (Mul e1 e2) = eval e1 * eval e2
eval (Div e1 e2) = eval e1 `div` eval e2
Making the evaluator safe (1)

safeEval :: Exp -> Maybe Integer
safeEval (Lit n) = Just n
safeEval (Add e1 e2) =
    case safeEval e1 of
      Nothing -> Nothing
      Just n1 ->
          case safeEval e2 of
            Nothing -> Nothing
            Just n2 -> Just (n1 + n2)
Making the evaluator safe (2)

safeEval (Sub e1 e2) =
    case safeEval e1 of
        Nothing -> Nothing
        Just n1 ->
            case safeEval e2 of
                Nothing -> Nothing
                Just n2 -> Just (n1 - n2)
Making the evaluator safe (3)

safeEval (Mul e1 e2) =
  case safeEval e1 of
    Nothing -> Nothing
    Just n1 ->
      case safeEval e2 of
        Nothing -> Nothing
        Just n2 -> Just (n1 * n2)
safeEval (Div e1 e2) = 
  case safeEval e1 of 
    Nothing -> Nothing 
    Just n1 -> 
      case safeEval e2 of 
        Nothing -> Nothing 
        Just n2 -> 
          if n2 == 0 
            then Nothing 
            else Just (n1 `div` n2)
Any common pattern?

Clearly a lot of code duplication!
Can we factor out a common pattern?
Any common pattern?

Clearly a lot of code duplication!
Can we factor out a common pattern?

We note:

- Sequencing of evaluations.
Any common pattern?

Clearly a lot of code duplication! Can we factor out a common pattern?

We note:

- Sequencing of evaluations.
- If one evaluation fail, fail overall.
Any common pattern?

Clearly a lot of code duplication! Can we factor out a common pattern?

We note:

- Sequencing of evaluations.
- If one evaluation fail, fail overall.
- Otherwise, make result available to following evaluations.
Sequencing evaluations (1)

```haskell
evalSeq :: Maybe Integer
         -> (Integer -> Maybe Integer)
         -> Maybe Integer

evalSeq ma f =
  case ma of
    Nothing  -> Nothing
    Just a   -> f a
```
Sequencing evaluations (2)

```haskell
safeEval :: Exp -> Maybe Integer
safeEval (Lit n) = Just n
safeEval (Add e1 e2) =
    safeEval e1 'evalSeq' (\n1 ->
    safeEval e2 'evalSeq' (\n2 ->
        Just (n1 + n2)))
safeEval (Sub e1 e2) =
    safeEval e1 'evalSeq' (\n1 ->
    safeEval e2 'evalSeq' (\n2 ->
        Just (n1 - n2)))
```
Sequencing evaluations (3)

```haskell
safeEval (Mul e1 e2) =
    safeEval e1 'evalSeq' (\n1 ->
    safeEval e2 'evalSeq' (\n2 ->
        Just (n1 - n2)))

safeEval (Div e1 e2) =
    safeEval e1 'evalSeq' (\n1 ->
    safeEval e2 'evalSeq' (\n2 ->
        if n2 == 0
        then Nothing
        else Just (n1 `div` n2)))
```
Aside: Scope rules of $\lambda$-abstractions

The scope rules of $\lambda$-abstractions are such that parentheses can be omitted:

```haskell
safeEval :: Exp -> Maybe Integer
...

safeEval (Add e1 e2) =
  safeEval e1 'evalSeq' \n1 ->
  safeEval e2 'evalSeq' \n2 ->
  Just (n1 + n2)
...
```

Exercise 1: Inline evalSeq (1)

safeEval (Add e1 e2) =
  safeEval e1 `evalSeq` \n1 ->
  safeEval e2 `evalSeq` \n2 ->
  Just (n1 + n2)
Exercise 1: Inline `evalSeq` (1)

\[
safeEval\ (\text{Add}\ e1\ e2) =
\begin{align*}
safeEval\ e1\ &\text{`evalSeq` }\ n1 \rightarrow \\
safeEval\ e2\ &\text{`evalSeq` }\ n2 \rightarrow \\
\text{Just } (n1 + n2)
\end{align*}
\]

= 

\[
safeEval\ (\text{Add}\ e1\ e2) =
\begin{align*}
\text{case } (\text{safeEval}\ e1)\ &\text{ of} \\
\text{Nothing } &\rightarrow \text{ Nothing} \\
\text{Just } a &\rightarrow (\ n1 \rightarrow \text{safeEval}\ e2 \ ...) \ a
\end{align*}
\]
Exercise 1: Inline evalSeq (2)

= 

safeEval (Add e1 e2) =
    case (safeEval e1) of
      Nothing -> Nothing
      Just n1 -> case safeEval e2 of
                    Nothing -> Nothing
                    Just a -> (\n2 -> ...) a
Exercise 1: Inline evalSeq (2)

safeEval (Add e1 e2) =
    case (safeEval e1) of
        Nothing -> Nothing
        Just n1 -> safeEval e2 'evalSeq' (\n2 -> ...)

= 

safeEval (Add e1 e2) =
    case (safeEval e1) of
        Nothing -> Nothing
        Just n1 -> case safeEval e2 of
                        Nothing -> Nothing
                        Just a -> (\n2 -> ...) a
Exercise 1: Inline evalSeq (3)

= 

safeEval (Add e1 e2) =
  case (safeEval e1) of
    Nothing -> Nothing
    Just n1 -> case safeEval e2 of
      Nothing -> Nothing
      Just n2 -> (Just n1 + n2)
**Maybe viewed as a computation (1)**

- Consider a value of type `Maybe a` as denoting a *computation* of a value of type `a` that *may fail*. 
Maybe viewed as a computation (1)

- Consider a value of type `Maybe a` as denoting a `computation` of a value of type `a` that `may fail`.

- When sequencing possibly failing computations, a natural choice is to fail overall once a subcomputation fails.
Maybe viewed as a computation (1)

- Consider a value of type `Maybe a` as denoting a computation of a value of type `a` that `may fail`.
- When sequencing possibly failing computations, a natural choice is to fail overall once a subcomputation fails.
- I.e. *failure is an effect*, implicitly affecting subsequent computations.
Maybe viewed as a computation (1)

- Consider a value of type `Maybe a` as denoting a *computation* of a value of type `a` that *may fail*.

- When sequencing possibly failing computations, a natural choice is to fail overall once a subcomputation fails.

- I.e. *failure is an effect*, implicitly affecting subsequent computations.

- Let’s generalize and adopt names reflecting our intentions.
Maybe viewed as a computation (2)

Successful computation of a value:

\[
\text{mbReturn} :: a \rightarrow \text{Maybe } a \\
\text{mbReturn} = \text{Just}
\]

Sequencing of possibly failing computations:

\[
\text{mbSeq} :: \text{Maybe } a \rightarrow (a \rightarrow \text{Maybe } b) \rightarrow \text{Maybe } b \\
\text{mbSeq } ma \ f = \\
\quad \text{case } ma \ \text{of} \\
\quad \quad \text{Nothing } \rightarrow \text{Nothing} \\
\quad \quad \text{Just } a \rightarrow f \ a
\]
Maybe viewed as a computation (3)

Failing computation:

```haskell
mbFail :: Maybe a
mbFail = Nothing
```
The safe evaluator revisited

safeEval :: Exp -> Maybe Integer

safeEval (Lit n) = mbReturn n

safeEval (Add e1 e2) =
    safeEval e1 `mbSeq` \n1 ->
    safeEval e2 `mbSeq` \n2 ->
    mbReturn (n1 + n2)

...

safeEval (Div e1 e2) =
    safeEval e1 `mbSeq` \n1 ->
    safeEval e2 `mbSeq` \n2 ->
    if n2 == 0 then mbFail
    else mbReturn (n1 `div` n2))
Example: Numbering trees

data Tree a = Leaf a | Tree a :^: Tree a

numberTree :: Tree a -> Tree Int
numberTree t = fst (ntAux t 0)
  where
    ntAux (Leaf _) n = (Leaf n, n+1)
    ntAux (t1 :^: t2) n =
      let (t1', n') = ntAux t1 n
      in let (t2', n'') = ntAux t2 n'
           in (t1' :^: t2', n'')
Observations

- Repetitive pattern: threading a counter through a sequence of tree numbering computations.
Observations

- Repetitive pattern: threading a counter through a *sequence* of tree numbering *computations*.
- It is very easy to pass on the wrong version of the counter!
Observations

- Repetitive pattern: threading a counter through a sequence of tree numbering computations.
- It is very easy to pass on the wrong version of the counter!

Can we do better?
A stateful computation consumes a state and returns a result along with a possibly updated state.
A _stateful computation_ consumes a state and returns a result along with a possibly updated state.

The following type synonym captures this idea:

```haskell
type S a = Int -> (a, Int)
```

(Only `Int` state for the sake of simplicity.)
A *stateful computation* consumes a state and returns a result along with a possibly updated state.

The following type synonym captures this idea:

\[
\text{type } S \ a = \text{Int} \rightarrow (a, \text{Int})
\]

(Only \text{Int} state for the sake of simplicity.)

A value (function) of type \( S \ a \) can now be viewed as denoting a stateful computation computing a value of type \( a \).
When sequencing stateful computations, the resulting state should be passed on to the next computation.
When sequencing stateful computations, the resulting state should be passed on to the next computation.

I.e. \textit{state updating is an effect}, implicitly affecting subsequent computations. (As we would expect.)
Stateful Computations (3)

Computation of a value without changing the state:

\[ \text{sReturn} :: a \rightarrow S a \]
\[ \text{sReturn} \ a = \ \lambda n \rightarrow (a, n) \]

Sequencing of stateful computations:

\[ \text{sSeq} :: S a \rightarrow (a \rightarrow S b) \rightarrow S b \]
\[ \text{sSeq} \ sa \ f = \ \lambda n \rightarrow \]
\[ \quad \text{let} \ (a, n') = sa \ n \]
\[ \quad \text{in} \ f \ a \ n' \]
Stateful Computations (4)

Reading and incrementing the state:

\[
s\text{Inc} :: S \text{ Int} \\
s\text{Inc} = \ n \rightarrow (n, n + 1)
\]
Numbering trees revisited

data Tree a = Leaf a | Tree a :^: Tree a

numberTree :: Tree a -> Tree Int
numberTree t = fst (ntAux t 0)
where
    ntAux (Leaf _) =
        sInc `sSeq` \n -> sReturn (Leaf n)
    ntAux (t1 :^: t2) =
        ntAux t1 `sSeq` \t1' ->
        ntAux t2 `sSeq` \t2' ->
        sReturn (t1' :^: t2')
Observations

- The “plumbing” has been captured by the abstractions.
Observations

• The “plumbing” has been captured by the abstractions.

• In particular, there is no longer any risk of “passing on” the wrong version of the state!
Comparison of the examples

- Both examples characterized by sequencing of effectful computations.
Comparison of the examples

- Both examples characterized by sequencing of effectful computations.
- Both examples could be neatly structured by introducing identically structured abstractions that encapsulated the effects:

  - A type denoting computations
  - A combinator for computing a value without any effect
  - A combinator for sequencing computations

In fact, both examples are instances of the general notion of a MONAD.
Comparison of the examples

- Both examples characterized by sequencing of effectful computations.
- Both examples could be neatly structured by introducing identically structured abstractions that encapsulated the effects:
  - A type denoting computations
Comparison of the examples

- Both examples characterized by sequencing of effectful computations.
- Both examples could be neatly structured by introducing identically structured abstractions that encapsulated the effects:
  - A type denoting computations
  - A combinator for computing a value without any effect
Comparison of the examples

- Both examples characterized by sequencing of effectful computations.
- Both examples could be neatly structured by introducing identically structured abstractions that encapsulated the effects:
  - A type denoting computations
  - A combinator for computing a value without any effect
  - A combinator for sequencing computations
Comparison of the examples

- Both examples characterized by sequencing of effectful computations.
- Both examples could be neatly structured by introducing identically structured abstractions that encapsulated the effects:
  - A type denoting computations
  - A combinator for computing a value without any effect
  - A combinator for sequencing computations
- In fact, both examples are instances of the general notion of a **MONAD**.
Monads in Functional Programming

A monad is represented by:

- A type constructor
  \[ \text{M} :: * \rightarrow * \]
  \( \text{M} \ T \) represents computations of a value of type \( \text{T} \).

- A polymorphic function
  \[ \text{return} :: a \rightarrow \text{M} \ a \]
  for lifting a value to a computation.

- A polymorphic function
  \[ (\gg\gg=) :: \text{M} \ a \rightarrow (a \rightarrow \text{M} \ b) \rightarrow \text{M} \ b \]
  for sequencing computations.
Exercise 2: join and fmap

Equivalently, the notion of a monad can be captured through the following functions:

\[
\begin{align*}
\text{return} & : \ a \rightarrow M \ a \\
\text{join} & : \ (M \ (M \ a)) \rightarrow M \ a \\
\text{fmap} & : \ (a \rightarrow b) \rightarrow (M \ a \rightarrow M \ b)
\end{align*}
\]

\text{join} “flattens” a computation, \text{fmap} “lifts” a function to map computations to computations.

Define \text{join} and \text{fmap} in terms of \text{>>=} (and \text{return}), and \text{>>=} in terms of \text{join} and \text{fmap}.
Exercise 2: Solution

\[ \text{join} :: \mathbb{M} (\mathbb{M} \ a) \rightarrow \mathbb{M} \ a \]
\[ \text{join } mm = mm >>= \text{id} \]

\[ \text{fmap} :: (a \rightarrow b) \rightarrow \mathbb{M} \ a \rightarrow \mathbb{M} \ b \]
\[ \text{fmap } f \ m = m >>= \ \lambda x \rightarrow \text{return} \ (f \ x) \]

\[ (\ggg) :: \mathbb{M} \ a \rightarrow (a \rightarrow \mathbb{M} \ b) \rightarrow \mathbb{M} \ b \]
\[ m >>= f = \text{join} \ (\text{fmap } f \ m) \]
Monad laws

Additionally, some simple laws must be satisfied:

\[
\begin{align*}
\text{return } x \mathbin{>>=} f &= f \, x \\
\text{m } \mathbin{>>=} \text{return } &= \text{m} \\
(\text{m } \mathbin{>>=} f) \mathbin{>>=} g &= \text{m } \mathbin{>>=} (\lambda x \rightarrow f \, x \mathbin{>>=} g)
\end{align*}
\]

I.e., \text{return} is the right and left identity for \(\mathbin{>>=}\),
and \(\mathbin{>>=}\) is associative.
Exercise 3: The Identity Monad

The *Identity Monad* can be understood as representing *effect-free* computations:

```
type I a = a
```

1. Provide suitable definitions of `return` and `>>=`.
2. Verify that the monad laws hold for your definitions.
Exercise 3: Solution

\[\text{return} :: \text{a} \to \text{I} \text{a}\]
\[\text{return} = \text{id}\]

\[\text{(>>=} :: \text{I} \text{a} \to (\text{a} \to \text{I} \text{b}) \to \text{I} \text{b}\]
\[\text{m >>=} \text{f} = \text{f m}\]
\[\text{-- or: (>>=} = \text{flip ($)$}}\]

Simple calculations verify the laws, e.g.:

\[\text{return} \ x \ >>=} \text{f} \ = \ \text{id} \ x \ >>=} \text{f}\]
\[\ = \ x \ >>=} \text{f}\]
\[\ = \ \text{f} \ x\]
The notion of a monad originated in Category Theory. There are several equivalent definitions (Benton, Hughes, Moggi 2000):

- **Kleisli triple/triple in extension form:** Most closely related to the >>= version:

  A *Klesili triple* over a category $\mathcal{C}$ is a triple $(T, \eta, \_^\ast)$, where $T : |\mathcal{C}| \rightarrow |\mathcal{C}|$, $\eta_A : A \rightarrow TA$ for $A \in |\mathcal{C}|$, $f^\ast : TA \rightarrow TB$ for $f : A \rightarrow TB$.

  (Additionally, some laws must be satisfied.)
Monads in Category Theory (2)

- **Monad/triple in monoid form**: More akin to the `join/fmap` version:

  A **monad** over a category \( C \) is a triple \((T, \eta, \mu)\), where \( T : C \to C \) is a functor, \( \eta : \text{id}_C \to T \) and \( \mu : T^2 \to T \) are natural transformations.

  (Additionally, some commuting diagrams must be satisfied.)
Monads in Haskell (1)

In Haskell, the notion of a monad is captured by a *Type Class*:

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

This allows the names of the common functions to be overloaded, and the sharing of derived definitions.
The Haskell monad class have two further methods with default instances:

\[
(\gg) \quad :: \quad m \ a \ \rightarrow \ m \ b \ \rightarrow \ m \ b \\
\quad m \gg k = m \gg\gg \_ \rightarrow k
\]

\[
\text{fail} \quad :: \quad \text{String} \ \rightarrow \ m \ a \\
\quad \text{fail} \ s = \text{error} \ s
\]
The **Maybe** monad in Haskell

```haskell
instance Monad Maybe where
    -- return :: a -> Maybe a
    return = Just

    -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
    Nothing >>= _ = Nothing
    (Just x) >>= f = f x
```
Exercise 4: A state monad in Haskell

Haskell 98 does not permit type synonyms to be instances of classes. Hence we have to define a new type:

```haskell
nenotype S a = S (Int -> (a, Int))

unS :: S a -> (Int -> (a, Int))
unS (S f) = f
```

Provide a Monad instance for `S`. 


instance Monad S where
    return a = S (\s -> (a, s))

    m >>= f = S $ \s ->
        let (a, s') = unS m s
        in unS (f a) s'
Monad-specific operations (1)

To be useful, monads need to be equipped with additional operations specific to the effects in question. For example:

\[
\text{fail} :: \text{String} \rightarrow \text{Maybe} \ a \\
\text{fail} \ s = \text{Nothing}
\]

\[
\text{catch} :: \text{Maybe} \ a \rightarrow \text{Maybe} \ a \rightarrow \text{Maybe} \ a \\
\text{m1} \ 'catch' \ \text{m2} = \\
\text{case} \ \text{m1} \ \text{of} \\
\text{Just} _ \rightarrow \text{m1} \\
\text{Nothing} \rightarrow \text{m2}
\]
Typical operations on a state monad:

\[
\begin{align*}
\text{set} & : \text{Int} \rightarrow S () \\
\text{set } a & = S (_{-} \rightarrow ((), a)) \\
\text{get} & : S \text{ Int} \\
\text{get} & = S (_{s} \rightarrow (s, s))
\end{align*}
\]

Moreover, there is often a need to “run” a computation. E.g.:

\[
\begin{align*}
\text{runS} & : S a \rightarrow a \\
\text{runS } m & = \text{fst} (\text{unS } m 0)
\end{align*}
\]
The do-notation (1)

Haskell provides convenient syntax for programming with monads:

```haskell
do
  a <- exp₁
  b <- exp₂
  return exp₃
```

is syntactic sugar for

```haskell
exp₁ >>= \a ->
exp₂ >>= \b ->
return exp₃
```
The **do-notation** (2)

Computations can be done solely for effect, ignoring the computed value:

```
do
  exp₁
  exp₂
  return exp₃
```

is syntactic sugar for

```
exp₁  >>= \_  ->
exp₂  >>= \_  ->
return exp₃
```
**The do-notation (3)**

A `let`-construct is also provided:

```plaintext
do
  let a = \text{expr}_1
  b = \text{expr}_2
  return \text{expr}_3
```

is equivalent to

```plaintext
do
  a <- return \text{expr}_1
  b <- return \text{expr}_2
  return \text{expr}_3
```
numberTree :: Tree a -> Tree Int

numberTree t = runS (ntAux t)

where

ntAux (Leaf _) = do
    n <- get
    set (n + 1)
    return (Leaf n)

ntAux (t1 :^: t2) = do
    t1' <- ntAux t1
    t2' <- ntAux t2
    return (t1' :^: t2')
Monadic utility functions

Some monad utilities, some from the Prelude, some from the module Monad:

\[
\begin{align*}
\text{sequence} & \quad : \quad \text{Monad } m \Rightarrow [m \ a] \rightarrow m \ [a] \\
\text{sequence\_} & \quad : \quad \text{Monad } m \Rightarrow [m \ a] \rightarrow m \ () \\
\text{mapM} & \quad : \quad \text{Monad } m \Rightarrow (a \rightarrow m \ b) \rightarrow [a] \rightarrow m \ [b] \\
\text{mapM\_} & \quad : \quad \text{Monad } m \Rightarrow (a \rightarrow m \ b) \rightarrow [a] \rightarrow m \ () \\
\text{when} & \quad : \quad \text{Monad } m \Rightarrow \text{Bool} \rightarrow m \ () \rightarrow m \ () \\
\text{foldM} & \quad : \quad \text{Monad } m \Rightarrow \ (a \rightarrow b \rightarrow m \ a) \rightarrow a \rightarrow [b] \rightarrow m \ a \\
\text{liftM} & \quad : \quad \text{Monad } m \Rightarrow (a \rightarrow b) \rightarrow (m \ a \rightarrow m \ b)
\end{align*}
\]
Exercise 5: Monadic utilities

Define

\[
\text{when} \quad :: \quad \text{Monad } m \Rightarrow \text{Bool} \rightarrow m () \rightarrow m ()
\]

\[
\text{sequence} \quad :: \quad \text{Monad } m \Rightarrow [m a] \rightarrow m [a]
\]

\[
\text{mapM} \quad :: \quad \text{Monad } m \Rightarrow (a \rightarrow m b) \rightarrow [a] \rightarrow m [b]
\]

in terms of the basic monad functions.
Exercise 5: Solution (1)

\[
\text{when} :: \text{Monad } m \Rightarrow \text{Bool} \rightarrow m () \rightarrow m () \\
\text{when } p \; m = \text{if } p \text{ then } m \text{ else return } ()
\]

\[
\text{sequence} :: \text{Monad } m \Rightarrow [m a] \rightarrow m [a] \\
\text{sequence } [] = \text{return } [] \\
\text{sequence } (ma : mas) = ma >>= \lambda a \rightarrow \\
\quad \text{sequence } mas >>= \lambda as \rightarrow \\
\quad \text{return } (a : as)
\]
Exercise 5: Solution (2)

\[
\text{mapM} :: \text{Monad m} \Rightarrow (a \rightarrow m b) \rightarrow [a] \rightarrow m [b]
\]

\[
\text{mapM } f \ [\] = \text{return } [\]
\]

\[
\text{mapM } f \ (a:as) = f \ a \ \gg= \ \backslash b \rightarrow \\
\quad \text{mapM } f \ as \ \gg= \ \backslash bs \rightarrow \\
\quad \text{return } (b:bs)
\]
In Haskell, IO is handled through the IO monad. IO is **abstract**! Conceptually:

\[
\text{newtype IO } a \equiv \text{IO (World } \to (a, \text{ World}))
\]

Some operations:

- `putChar :: Char -> IO ()`
- `putStr :: String -> IO ()`
- `putStrLn :: String -> IO ()`
- `getChar :: IO Char`
- `getLine :: IO String`
- `getContents :: String`
The ST Monad: “real” state

The ST monad (common Haskell extension) provides real, imperative state behind the scenes to allow efficient implementation of imperative algorithms:

```haskell
data ST s a -- abstract
instance Monad (ST s)

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

runST :: (forall s . st s a) -> a
```
Nondeterminism: The list monad

instance Monad [] where
  return a = [a]
  m >>= f = concat (map f m)
  fail s = []

Example:

do
  x <- [1, 2]
  y <- ['a', 'b']
  return (x,y)

Result: [(1,'a'),(1,'b'),(2,'a'),(2,'b')]
instance Monad ((->) e) where
    return a = const a
    m >>= f = \e -> f (m e) e

getEnv :: ((->) e) e
getEnv = id

Cf. the combinators S, K, and I!

I :: a -> a
K :: a -> b -> a
S :: (a -> b -> c) -> (a -> b) -> a -> c
(>>>=) :: (a -> b) -> (b -> a -> c) -> a -> c
In Continuation-Passing style (CPS), a continuation representing the “rest of the computation” is passed to each computation.
The continuation monad (1)

- In Continuation-Passing style (CPS), a *continuation* representing the “rest of the computation” is passed to each computation.
- A continuation is a function that when applied to the result of the current subcomputation, returns the final result of the overall computation.
The continuation monad (1)

- In Continuation-Passing style (CPS), a \textit{continuation} representing the “rest of the computation” is passed to each computation.

- A continuation is a function that when applied to the result of the current subcomputation, returns the final result of the overall computation.

- Making continuations explicitly available makes it possible to implement control-flow effects, like jumps.
data CPS r a = CPS ((a -> r) -> r)

unCPS :: CPS r a -> ((a -> r) -> r)
unCPS (CPS f) = f

instance Monad (CPS r) where
    return a = CPS (\k -> k a)
    m >>= f = CPS $ \k ->
        unCPS m (\a -> unCPS (f a) k)
The continuation monad (3)

\[
callCC :: ((a -> CPS r b) -> CPS r a) -> CPS r a
\]

\[
callCC f = CPS \$ \k ->
  unCPS (f (\a -> CPS (\_ -> k a))) k
\]

\[
runCPS :: CPS a a -> a
\]

\[
runCPS m = unCPS m id
\]
Exercise 6: Control transfer

\[ f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[ f x y = \text{runCPS} \$ \ do \]
\[ \quad \text{callCC} \$ \ \backslash \text{exit} \rightarrow \ do \]
\[ \quad \quad \text{let} \ d = x - y \]
\[ \quad \quad \text{when} \ (d == 0) \ (\text{exit} \ (-1)) \]
\[ \quad \quad \text{let} \ z = (\text{abs} \ ((x + y) \ 'div' \ d)) \]
\[ \quad \quad \text{when} \ (z > 10) \ (\text{exit} \ (-2)) \]
\[ \quad \text{return} \ (z^3) \]

Compute \( f 10 6, f 10 10, \text{and} f 10 9. \)
A Thread represents a process: a stream of primitive atomic operations:

```
data Thread = Print Char Thread
             | Fork Thread Thread
             | End
```
A Thread represents a process: a stream of primitive *atomic* operations:

```
data Thread = Print Char Thread
              | Fork Thread Thread
              | End
```

Note that a Thread represents the *entire rest* of a computation.
Introduce a monad representing “ interleavable computations”. At this stage, this amounts to little more than a convenient way to construct threads by sequential composition.
Introduce a monad representing “interleavable computations”. At this stage, this amounts to little more than a convenient way to construct threads by sequential composition.

How can Threads be composed sequentially? The only way is to parameterize thread prefixes on the rest of the Thread. This leads directly to continuations.
newtype CM a = CM ((a -> Thread) -> Thread)

fromCM :: CM a -> ((a -> Thread) -> Thread)
fromCM (CM x) = x

thread :: CM a -> Thread
thread m = fromCM m (const End)

instance Monad CM where
    return x = CM ($ k -> k x)
    m >>= f = CM $ \k ->
        fromCM m ($ x -> fromCM (f x) k)
A Concurrency Monad (4)

Atomic operations:

\[
\begin{align*}
c\text{Print} & : \text{Char} \to \text{CM} () \\
c\text{Print} \ c & = \text{CM} (\ k \to \text{Print} \ c \ (k ()) ) \\

c\text{Fork} & : \text{CM} \ a \to \text{CM} () \\
c\text{Fork} \ m & = \text{CM} (\ k \to \text{Fork} \ (\text{thread} \ m) \ (k ()) ) \\

c\text{End} & : \text{CM} \ a \\
c\text{End} & = \text{CM} (\ _\to \text{End})
\end{align*}
\]
A Concurrency Monad (5)

Running a computation:

define Output = [Char]
define ThreadQueue = [Thread]
define State = (Output, ThreadQueue)

runCM :: CM a -> Output
runCM m = runHlp ("", []) (thread m)
where

runHlp s t =
    case dispatch s t of
        Left (s', t) -> runHlp s' t
        Right o     -> o
A Concurrency Monad (6)

Dispatch on the operation of the currently running Thread. Then call the scheduler.

\[
\text{dispatch} :: \text{State} \rightarrow \text{Thread} \\
\rightarrow \text{Either (State, Thread) Output}
\]

\[
\text{dispatch} \ (o, \ rq) \ \text{(Print} \ c \ \text{t}) = \\
\text{schedule} \ (o \ +\ [c], \ rq \ +\ [t])
\]

\[
\text{dispatch} \ (o, \ rq) \ \text{(Fork} \ t1 \ \text{t2}) = \\
\text{schedule} \ (o, \ rq \ +\ [t1, \ t2])
\]

\[
\text{dispatch} \ (o, \ rq) \ \text{End} = \\
\text{schedule} \ (o, \ rq)
\]
A Concurrency Monad (7)

Selects next Thread to run, if any.

\[
\begin{align*}
\text{schedule} & : \text{State} \rightarrow \text{Either (State, Thread)} \\
\text{schedule} \ (o, \ []) & = \text{Right} \ o \\
\text{schedule} \ (o, \ t:ts) & = \text{Left} \ ((o, \ ts), \ t)
\end{align*}
\]
Example: Concurrent processes

\[
\begin{align*}
\text{p1} &:: \text{CM} () \\
\text{p1} &= \text{do} \\
&\quad \text{cPrint 'a'} \\
&\quad \text{cPrint 'b'} \\
&\quad \ldots \\
&\quad \text{cPrint 'j'} \\
\text{p2} &:: \text{CM} () \\
\text{p2} &= \text{do} \\
&\quad \text{cPrint '1'} \\
&\quad \text{cPrint '2'} \\
&\quad \ldots \\
&\quad \text{cPrint '0'} \\
\text{p3} &:: \text{CM} () \\
\text{p3} &= \text{do} \\
&\quad \text{cFork p1} \\
&\quad \text{cFork p2} \\
&\quad \text{cPrint 'A'} \\
&\quad \text{cPrint 'B'}
\end{align*}
\]

main = print (runCM p3)

Result: aAbc1Bd2e3f4g5h6i7j890
(As it stands, the output is only made available after \textit{all} threads have terminated.)
Alternative version

Incremental output:

runCM :: CM a -> Output
runCM m = dispatch [] (thread m)

dispatch :: ThreadQueue -> Thread -> Output
dispatch rq (Print c t) = c : schedule (rq ++ [t])
dispatch rq (Fork t1 t2) = schedule (rq ++ [t1, t2])
dispatch rq End = schedule rq

schedule :: ThreadQueue -> Output
schedule [] = []
schedule (t:ts) = dispatch ts t
Example: Concurrent processes 2

\[
\begin{align*}
\text{p1} &:: \text{CM ()} & \text{p2} &:: \text{CM ()} & \text{p3} &:: \text{CM ()} \\
\text{p1} &= \text{do} & \text{p2} &= \text{do} & \text{p3} &= \text{do} \\
&\quad \text{cPrint 'a'} & &\quad \text{cPrint '1'} & &\quad \text{cFork} \ p1 \\
&\quad \text{cPrint 'b'} & &\quad \text{undefined} & &\quad \text{cPrint 'A'} \\
&\quad \ldots & &\quad \ldots & &\quad \text{cFork} \ p2 \\
&\quad \text{cPrint 'j'} & &\quad \text{cPrint '0'} & &\quad \text{cPrint 'B'}
\end{align*}
\]

\[
\text{main} = \text{print} \ (\text{runCM} \ \text{p3})
\]

**Result:** aAbc1Bd*** Exception: Prelude.undefined
Reading

- Nomaware. *All About Monads.*
  http://www.nomaware.com/monads

