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.
- 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 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

```haskell
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)

```haskell
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)

```haskell
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)

```haskell
safeEval (Mul e1 e2) =
  case safeEval e1 of
    Nothing -> Nothing
    Just n1 ->
      case safeEval e2 of
        Nothing -> Nothing
        Just n2 -> Just (n1 * n2)
```

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.

Making the evaluator safe (4)

```haskell
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)
```

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))
    )
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)
...
```

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)

Exercise 1: Inline evalSeq (1)

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

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

```haskell
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**.
- 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.

---

**Exercise 1: Inline evalSeq (3)**

```haskell
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 (2)**

**Successful computation of a value:**

```haskell
mbReturn :: a -> Maybe a
mbReturn = Just
```

**Sequencing of possibly failing computations:**

```haskell
mbSeq :: Maybe a -> (a -> Maybe b) -> Maybe b
mbSeq ma f =
  case ma of
    Nothing -> Nothing
    Just a -> f a
```
Maybe viewed as a computation (3)

Failing computation:

\[
\text{mbFail :: Maybe a} \\
\text{mbFail = Nothing}
\]

Example: Numbering trees

\[
data \text{ Tree a = Leaf a} \ |
\text{ Tree a :}^*: \text{ Tree a}
\]

\[
\text{numberTree :: Tree a \to Tree Int} \\
\text{numberTree t = fst (ntAux t 0)}
\]

where

\[
\begin{align*}
\text{ntAux (Leaf _)} & \quad \text{n = (Leaf n, n+1)} \\
\text{ntAux (t1 :^*: t2)} & \quad \text{n} = \\
& \quad \text{let (t1’, n’) = ntAux t1 n} \\
& \quad \text{in let (t2’, n”’) = ntAux t2 n’} \\
& \quad \text{in (t1’ :^*: t2’, n”’)}
\end{align*}
\]

The safe evaluator revisited

\[
\text{safeEval :: Exp \to Maybe Integer} \\
\text{safeEval (Lit n) = mbReturn n} \\
\text{safeEval (Add e1 e2) =} \\
\quad \text{safeEval e1 ‘mbSeq’ } \backslash \text{n1 \to} \\
\quad \text{safeEval e2 ‘mbSeq’ } \backslash \text{n2 \to} \\
\quad \text{mbReturn (n1 + n2)}
\]

\[
\ldots\text{safeEval (Div e1 e2) =} \\
\quad \text{safeEval e1 ‘mbSeq’ } \backslash \text{n1 \to} \\
\quad \text{safeEval e2 ‘mbSeq’ } \backslash \text{n2 \to} \\
\quad \text{if n2 == 0 then mbFail} \\
\quad \text{else mbReturn (n1 ‘div’ n2))}
\]

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?
**Stateful Computations (1)**

- 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 value (function) of type \( S a \) can now be viewed as denoting a stateful computation computing a value of type \( a \).

**Stateful Computations (2)**

- When sequencing stateful computations, the resulting state should be passed on to the next computation.
- I.e. *state updating is an effect*, implicitly affecting subsequent computations. (As we would expect.)

**Stateful Computations (3)**

Computation of a value without changing the state:

```haskell
sReturn :: a -> S a
sReturn a = \n -> (a, n)
```

Sequencing of stateful computations:

```haskell
sSeq :: S a -> (a -> S b) -> S b
sSeq sa f = \n ->
  let (a, n') = sa n
  in f a n'
```

**Stateful Computations (4)**

Reading and incrementing the state:

```haskell
sInc :: S Int
sInc = \n -> (n, n + 1)
```
Numbering trees revisited

```
data Tree a = Leaf a | Tree a :ˆ: Tree a

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

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.
- 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
  \[ M :: * -> * \]
  \[ M T \] represents computations of a value of type \( T \).
- A polymorphic function
  \[ \text{return} :: a -> M a \]
  for lifting a value to a computation.
- A polymorphic function
  \[ (>>=) :: M a -> (a -> M b) -> M b \]
  for sequencing computations.
**Exercise 2: join and fmap**

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

```haskell
return :: a -> M a
join :: (M (M a)) -> M a
fmap :: (a -> b) -> (M a -> M b)
```

join “flattens” a computation, fmap “lifts” a function to map computations to computations.

Define join and fmap in terms of >>= (and return), and >>= in terms of join and fmap.

**Monad laws**

Additionally, some simple laws must be satisfied:

```haskell
return x >>= f = f x
m >>= return = m
(m >>= f) >>= g = m >>= (\x -> f x >>= g)
```

I.e., return is the right and left identity for >>=, and >>= is associative.

**Exercise 2: Solution**

```haskell
join :: M (M a) -> M a
join mm = mm >>= id

fmap :: (a -> b) -> M a -> M b
fmap f m = m >>= \x -> return (f x)

(>>=) :: M a -> (a -> M b) -> M b
m >>= f = join (fmap f m)
```

**Exercise 3: The Identity Monad**

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

```haskell
type I a = a
```

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

return :: a -> I a
return = id

(>>=) :: I a -> (a -> I b) -> I b
m >>= f = f m
-- or: (>>=) = flip ($)  

Simple calculations verify the laws, e.g.:

\[ \text{return } x \mathrel{ >>= } f = \text{id } x \mathrel{ >>= } f \]
\[ = x \mathrel{ >>= } f \]
\[ = f x \]

Monads in Category Theory (1)

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 *Kleisli triple* over a category \( C \) is a triple \((T, \eta, \mu)\), where \( T : |C| \to |C| \), \( \eta_A : A \to TA \) for \( A \in |C| \), \( f^* : TA \to TB \) for \( f : A \to TB \).

  (Additionally, some laws 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.

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 (2)

The Haskell monad class have two further methods with default instances:

\[
(>>): \text{m a} \to \text{m b} \to \text{m b}
\]
\[
m \gg k = m >>= \_ \to k
\]

\[
\text{fail} : \text{String} \to \text{m a}
\]
\[
\text{fail } s = \text{error } s
\]

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:

\[
\text{newtype } S \ a = S (\text{Int} \to (a, \text{Int}))
\]

\[
\text{unS} : S a \to (\text{Int} \to (a, \text{Int}))
\]
\[
\text{unS } (S f) = f
\]

Provide a Monad instance for \(S\).
Monad-specific operations (1)

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

```haskell
fail :: String -> Maybe a
fail s = Nothing

catch :: Maybe a -> Maybe a -> Maybe a
m1 'catch' m2 =
  case m1 of
    Just _ -> m1
    Nothing -> m2
```

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
```

Monad-specific operations (2)

Typical operations on a state monad:

```haskell
set :: Int -> S ()
set a = S (\_ -> ((), a))

get :: S Int
get = S (\s -> (s, s))
```

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

```haskell
runS :: S a -> a
runS m = fst (unS m 0)
```

The do-notation (2)

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

```haskell
do
  exp₁
  exp₂
  return exp₃
```

is syntactic sugar for

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

A let-construct is also provided:

\[
\begin{align*}
\text{do} & \quad \text{let } a = \text{exp}_1 \\
& \quad b = \text{exp}_2 \\
& \quad \text{return } \text{exp}_3
\end{align*}
\]

is equivalent to

\[
\begin{align*}
\text{do} & \quad a \leftarrow \text{return } \text{exp}_1 \\
& \quad b \leftarrow \text{return } \text{exp}_2 \\
& \quad \text{return } \text{exp}_3
\end{align*}
\]

Numbering trees in do-notation

numberTree :: Tree a -> Tree Int
numberTree t = runS (ntAux t)
where

\[
\begin{align*}
\text{ntAux } (\text{Leaf } _) & = \text{do} \\
& \quad n \leftarrow \text{get} \\
& \quad \text{set } (n + 1) \\
& \quad \text{return } (\text{Leaf } n)
\end{align*}
\]

\[
\begin{align*}
\text{ntAux } (t_1 :^\cdot: t_2) & = \text{do} \\
& \quad t_1' \leftarrow \text{ntAux } t_1 \\
& \quad t_2' \leftarrow \text{ntAux } t_2 \\
& \quad \text{return } (t_1' :^\cdot: t_2')
\end{align*}
\]

Monad utility functions

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

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

Exercise 5: Monadic utilities

Define

\[
\begin{align*}
\text{when} & \quad : \text{Monad } m \Rightarrow \text{Bool} \rightarrow m () \rightarrow m () \\
\text{sequence} & \quad : \text{Monad } m \Rightarrow [m \ a] \rightarrow m [a] \\
\text{mapM} & \quad : \text{Monad } m \Rightarrow (a \rightarrow m \ b) \rightarrow [a] \rightarrow m [b]
\end{align*}
\]

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

**when** :: Monad m => Bool -> m () -> m ()
when \( p \) \( m \) = if \( p \) then \( m \) else return ()

**sequence** :: Monad m => [m a] -> m [a]
sequence [] = return []
sequence (ma:mas) = ma >>= \( \lambda a \rightarrow \)
sequence mas >>= \( \lambda as \rightarrow \)
return (a:as)

### Exercise 5: Solution (2)

**mapM** :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f [] = return []
mapM f (a:as) = f a >>= \( \lambda b \rightarrow \)
mapM f as >>= \( \lambda bs \rightarrow \)
return (b:bs)

---

### The Haskell IO monad

In Haskell, IO is handled through the IO monad. IO is **abstract**! Conceptually:

```haskell
define newtype IO a = IO (World -> (a, 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
define data ST s a = ST (STRef s a)
define instance Monad (ST s)
define newSTRef :: s ST a (STRef s a)
define readSTRef :: STRef s a -> ST s a
define writeSTRef :: STRef s a -> a -> ST s ()
```

```haskell
define runST :: (forall s . ST s a) -> a
```
**Nondeterminism: The list monad**

```haskell
data [a] 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')]```

**Environments: The reader monad**

```haskell
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
```

**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.
- Making continuations explicitly available makes it possible to implement control-flow effects, like jumps.

**The continuation monad (2)**

```haskell
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 :: Int -> Int -> Int
f x y = runCPS $ do
  callCC $ \exit -> do
    let d = x - y
    when (d == 0) (exit (-1))
    let z = (abs ((x + y) `div` d))
    when (z > 10) (exit (-2))
    return (z^3)

Compute f 10 6, f 10 10, and f 10 9.

A Concurrency Monad (1)

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.

A Concurrency Monad (2)

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.
A Concurrency Monad (3)

```haskell
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:

```haskell
cPrint :: Char -> CM ()
cPrint c = CM (\k -> Print c (k ()))

cFork :: CM a -> CM ()
cFork m = CM (\k -> Fork (thread m) (k ()))

cEnd :: CM a
cEnd = CM (\_ -> End)
```

A Concurrency Monad (5)

Running a computation:

```haskell
running a computation:

type Output = [Char]
type ThreadQueue = [Thread]
type 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.

```haskell
dispatch :: State -> Thread
  -> Either (State, Thread) Output

dispatch (o, rq) (Print c t) =
  schedule (o ++ [c], rq ++ [t])
dispatch (o, rq) (Fork t1 t2) =
  schedule (o, rq ++ [t1, t2])
dispatch (o, rq) End =
  schedule (o, rq)
```
A Concurrency Monad (7)

Selects next Thread to run, if any.

```haskell
schedule :: State -> Either (State, Thread)
        Output
schedule (o, []) = Right o
schedule (o, t:ts) = Left ((o, ts), t)
```

Example: Concurrent processes

```haskell
p1 :: CM ()
p2 :: CM ()
p3 :: CM ()
p1 = do cPrint 'a'
cPrint '1'
cFork p1
cPrint 'b'
cPrint '2'
cFork p2
... ...
cPrint 'j'
cPrint '0'
cFork p3
cPrint 'B'
main = print (runCM p3)
Result: aAbc1Bd2e3f4g5h6i7j890
(As it stands, the output is only made available after all threads have terminated.)
```

Alternative version

Incremental output:

```haskell
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

```haskell
p1 :: CM ()
p2 :: CM ()
p3 :: CM ()
p1 = do cPrint 'a'
cPrint '1'
cFork p1
cPrint 'b'
undefined cPrint 'A'
... ...
cPrint 'j'
cPrint '0'
cPrint 'B'
main = print (runCM p3)
Result: aAbc1Bd2e3f4g5h6i7j890
(As it stands, the output is only made available after all threads have terminated.)
```
Reading

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

