Monads in Haskell

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
```

Allows names of the common functions to be overloaded and sharing of derived definitions.

This Lecture

- Monads in Haskell
- Some standard monads
- Combining effects: monad transformers
- Arrows
- FRP and Yampa

The Maybe Monad in Haskell

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

  -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  --         -> Maybe b
  Nothing >>= _ = Nothing
  (Just x) >>= f = f x
```
Exercise 1: 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
ewtype S a = S (Int -> (a, Int))
unS :: S a -> (Int -> (a, Int))
unS (S f) = f
```

Provide a Monad instance for `S`.

Exercise 1: Solution

```haskell
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:

```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
```

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, need to “run” a computation. E.g.:

```haskell
runS :: S a -> a
runS m = fst (unS m 0)
```
The do-notation (1)

Haskell provides convenient syntax for programming with monads:

```
do
  a <- exp_1
  b <- exp_2
  return exp_3
```

is syntactic sugar for

```
exp_1 >>= \a ->
exp_2 >>= \b ->
return exp_3
```

The do-notation (2)

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

```
do
  exp_1
  exp_2
  return exp_3
```

is syntactic sugar for

```
exp_1 >>= \_ ->
exp_2 >>= \_ ->
return exp_3
```

The do-notation (3)

A let-construct is also provided:

```
do
  let a = exp_1
     b = exp_2
  return exp_3
```

is equivalent to

```
do
  a <- return exp_1
  b <- return exp_2
  return exp_3
```

Numbering Trees in do-notation

```haskell
numberTree :: Tree a -> Tree Int
numberTree t = runS (ntAux t)
where
  ntAux :: Tree a -> S (Tree Int)
  ntAux (Leaf _) = do
    n <- get
    set (n + 1)
    return (Leaf n)
  ntAux (Node t1 t2) = do
    t1' <- ntAux t1
    t2' <- ntAux t2
    return (Node t1' t2')
```
Given a suitable “Diagnostics” monad \( D \) that collects error messages, \( \text{enterVar} \) can be turned from this:

\[
\text{enterVar} :: \text{Id} \to \text{Int} \to \text{Type} \to \text{Env} \to \text{Either Env ErrorMgs}
\]

into this:

\[
\text{enterVarD} :: \text{Id} \to \text{Int} \to \text{Type} \to \text{Env} \to D \text{ Env}
\]

and then \( \text{identDefs} \) from this ...

\[
\text{identDefsD} \ l \ \text{env} \ [\] = \text{return} \ ([][], \ \text{env})
\]

\[
\text{identDefsD} \ l \ \text{env} \ ((i,t,e) : ds) = \text{do}
\]

\[
e' \leftarrow \text{identAuxD} \ l \ \text{env} \ e
\]

\[
e'v' \leftarrow \text{enterVarD} \ i \ l \ t \ \text{env}
\]

\[
(ds'', \ env'') \leftarrow \text{identDefsD} \ l \ \text{env}' \ ds
\]

\[
\text{return} \ ((i,t,e') : ds'', \ env'')
\]

(Suffix \( D \) just to remind us the types have changed.)

\[
\text{identDefs} \ l \ \text{env} \ [\] = ([], \ \text{env})
\]

\[
\text{identDefs} \ l \ \text{env} \ ((i,t,e) : ds) = ((i,t,e') : ds', \ env'')
\]

\[
\text{where}
\]

\[
e' = \text{identAux} \ l \ \text{env} \ e
\]

\[
\text{env}' = \text{enterVar} \ i \ l \ t \ \text{env}
\]

\[
(ds', \ env'', \ ds) = \text{identDefs} \ l \ \text{env} \ ds
\]

The monadic version is very close to ideal, without sacrificing functionality, clarity, or pureness!
The List Monad

Computation with many possible results, "nondeterminism":

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

Example: Result:

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

The Reader Monad

Computation in an environment:

```
instance Monad ((->) e) where
    return a = const a
    m >>= f = \e -> f (m e) e

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

The Haskell IO Monad

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

```
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
```

Monad Transformers (1)

What if we need to support more than one type of effect?
For example: State and Error/Partiality?
We could implement a suitable monad from scratch:

```
newtype SE s a = SE (s -> Maybe (a, s))
```
Monad Transformers (2)

However:

- Not always obvious how: e.g., should the combination of state and error have been
  newtype SE s a = SE (s -> (Maybe a, s))
- Duplication of effort: similar patterns related to specific effects are going to be repeated over and over in the various combinations.

Monad Transformers (3)

Monad Transformers can help:

- A monad transformer transforms a monad by adding support for an additional effect.
- A library of monad transformers can be developed, each adding a specific effect (state, error, ...), allowing the programmer to mix and match.
- A form of aspect-oriented programming.

Monad Transformers in Haskell (1)

- A monad transformer maps monads to monads. Represented by a type constructor T of the following kind:
  \[ T :: (\ast \rightarrow \ast) \rightarrow (\ast \rightarrow \ast) \]
- Additionally, a monad transformer adds computational effects. A mapping lift from computations in the underlying monad to computations in the transformed monad is needed:
  \[ \text{lift} :: M a \rightarrow T M a \]

Monad Transformers in Haskell (2)

- These requirements are captured by the following (multi-parameter) type class:
  \[ \text{class } \ (\text{Monad m, Monad } (t \text{ m})) \rightarrow \text{MonadTransformer } t \text{ m where} \text{lift} :: m a \rightarrow t m a \]
Classes for Specific Effects

A monad transformer adds specific effects to any monad. Thus the effect-specific operations needs to be overloaded. For example:

```hs
class Monad m => E m where
eFail :: m a
eHandle :: m a -> m a -> m a
```

```hs
class Monad m => S m s | m -> s where
sSet :: s -> m ()
sGet :: m s
```

The Identity Monad

We are going to construct monads by successive transformations of the identity monad:

```hs
newtype I a = I a
unI (I a) = a

instance Monad I where
  return a = I a
  m >>= f = f (unI m)

runI :: I a -> a
runI = unI
```

The Error Monad Transformer (1)

```hs
newtype ET m a = ET (m (Maybe a))
unET (ET m) = m
```

Any monad transformed by ET is a monad:

```hs
instance Monad m => Monad (ET m) where
  return a = ET (return (Just a))
  m >>= f = ET $ do
    ma <- unET m
    case ma of
      Nothing -> return Nothing
      Just a -> unET (f a)
```

The Error Monad Transformer (2)

We need the ability to run transformed monads:

```hs
runET :: Monad m => ET m a -> m a
runET etm = do
  ma <- unET etm
  case ma of
    Just a -> return a
    Nothing -> error "Should not happen"
```

ET is a monad transformer:

```hs
instance Monad m => MonadTransformer ET m where
  lift m = ET (m >>= \a -> return (Just a))
```
Any monad transformed by \( ET \) is an instance of \( E \):

\[
\text{instance Monad } m \Rightarrow E \ (ET \ m) \text{ where}
\]
\[
eFail = ET \ (\text{return Nothing})
\]
\[
m_1 \ 'eHandle' \ m_2 = ET \ \&\&
\]
\[
ma \leftarrow \text{unET} \ m_1
\]
\[
case ma \ of
\]
\[
\text{Nothing} \rightarrow \text{unET} \ m_2
\]
\[
\text{Just } _\_ \rightarrow \text{return} \ ma
\]

A state monad transformed by \( ET \) is a state monad:

\[
\text{instance } S \ m \ s \Rightarrow S \ (ET \ m) \ s \text{ where}
\]
\[
sSet \ s = \text{lift} \ (sSet \ s)
\]
\[
sGet = \text{lift} \ sGet
\]

Let

\[
ex2 = \text{eFail} \ 'eHandle' \ \text{return} \ 1
\]

1. Suggest a possible type for \( ex2 \).
   (Assume \( 1 :: \text{Int} \).)

2. Given your type, use the appropriate combination of “run functions” to run \( ex2 \).

\[
ex2 :: ET \ I \ Int
\]
\[
ex2 = \text{eFail} \ 'eHandle' \ \text{return} \ 1
\]

\[
ex2result :: \text{Int}
\]
\[
ex2result = \text{runI} \ (\text{runET} \ ex2)
\]
The State Monad Transformer (1)

newtype ST s m a = ST (s -> m (a, s))
unST (ST m) = m

Any monad transformed by ST is a monad:
instance Monad m => Monad (ST s m) where
    return a = ST (\s -> return (a, s))
    m >>= f = ST \s -> do
        (a, s') <- unST m s
        unST (f a) s'

LiU-FP2010 Part II: Lecture 6 – p.33/83

The State Monad Transformer (2)

We need the ability to run transformed monads:
runST :: Monad m => ST s m a -> s -> m a
runST stf s0 = do
    (a, _) <- unST stf s0
    return a

ST is a monad transformer:
instance Monad m => MonadTransformer (ST s) m where
    lift m = ST (\s -> m >>= \a ->
        return (a, s))

LiU-FP2010 Part II: Lecture 6 – p.34/83

The State Monad Transformer (3)

Any monad transformed by ST is an instance of S:
instance Monad m => S (ST s m) s where
    sSet s = ST (\_ -> return (((), s))
    sGet = ST (\s -> return (s, s))

An error monad transformed by ST is an error monad:
instance E m => E (ST s m) where
    eFail = lift eFail
    m1 'eHandle' m2 = ST $ \s ->
        unST m1 s 'eHandle' unST m2 s

LiU-FP2010 Part II: Lecture 6 – p.35/83

Exercise 3: Effect Ordering

Consider the code fragment
ex3a :: (ST Int (ET I)) Int
ex3a = (sSet 42 >> eFail) 'eHandle' sGet

Note that the exact same code fragment also can be typed as follows:
ex3b :: (ET (ST Int I)) Int
ex3b = (sSet 42 >> eFail) 'eHandle' sGet

What is
runI (runET (runST ex3a 0))
runI (runST (runET ex3b) 0)

LiU-FP2010 Part II: Lecture 6 – p.36/83
**Exercise 3: Solution**

\[
\text{runI (runET (runST ex3a 0))} = 0 \\
\text{runI (runST (runET ex3b) 0)} = 42
\]

Why? Because:

\[
\begin{align*}
\text{ST s (ET I) a} & \rightsquigarrow s \rightarrow (ET I) (a, s) \\
& \rightarrow s \rightarrow I (\text{Maybe } (a, s)) \\
& \rightarrow s \rightarrow \text{Maybe } (a, s)
\end{align*}
\]

\[
\begin{align*}
\text{ET (ST s I) a} & \rightsquigarrow (ST s I) (\text{Maybe } a) \\
& \rightarrow s \rightarrow I (\text{Maybe } a, s) \\
& \rightarrow s \rightarrow (\text{Maybe } a, s)
\end{align*}
\]

**Exercise 4: Alternative ST?**

To think about.

Could ST have been defined in some other way, e.g.

\[
\text{newtype ST s m a = ST } (m \ (s \rightarrow (a, s)))
\]

or perhaps

\[
\text{newtype ST s m a = ST } (s \rightarrow (m \ a, s))
\]

**Problems with Monad Transformers**

- With one transformer for each possible effect, we get a lot of combinations: the number grows quadratically; each has to be instantiated explicitly.
- Jaskelioff (2008, 2009) has proposed a possible, more extensible alternative.

**Arrows (1)**

System descriptions in the form of block diagrams are very common. Blocks have inputs and outputs and can be combined into larger blocks. For example, serial composition:

\[
\begin{array}{c}
\text{f} \\
\end{array} \quad \begin{array}{c} \swarrow \\
\text{g} \end{array}
\]

A **combinator** can be defined that captures this idea:

\[
(\triangleright\triangleright\triangleright) \ : \ B \ a \ b \ \rightarrow \ B \ b \ c \ \rightarrow \ B \ a \ c
\]
Arrows (2)

But systems can be complex:

How many and what combinators do we need to be able to describe arbitrary systems?

Arrows (3)

John Hughes' **arrow** framework:

- Abstract data type interface for function-like types (or “blocks”, if you prefer).
- Particularly suitable for types representing process-like computations.
- Related to **monads**, since arrows are computations, but more general.
- Provides a minimal set of “wiring” combinators.

What is an arrow? (1)

- A **type constructor** \( a \) of arity two.
- Three operators:
  - **lifting**:
    \[ \text{arr} :: (b \to c) \to a\ b\ c \]
  - **composition**:
    \[ (\ggg) :: a\ b\ c \to a\ c\ d \to a\ b\ d \]
  - **widening**:
    \[ \text{first} :: a\ b\ c \to a\ (b,d)\ (c,d) \]
- A set of **algebraic laws** that must hold.

What is an arrow? (2)

These diagrams convey the general idea:

\[ \begin{align*}
\text{arr}\ f & \quad f \ggg g \\
\text{first}\ f &
\end{align*} \]
The Arrow class

In Haskell, a type class is used to capture these ideas (except for the laws):

```haskell
class Arrow a where
  arr :: (b -> c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b,d) (c,d)
```

Functions are arrows (1)

Functions are a simple example of arrows, with (->) as the arrow type constructor.

**Exercise 5:** Suggest suitable definitions of

- `arr`
- `(>>>)`
- `first`

for this case!

(We have not looked at what the laws are yet, but they are “natural”.)

Functions are arrows (2)

Solution:

- `arr = id`
  
  To see this, recall
  
  ```haskell
  id :: t -> t
  arr :: (b->c) -> a b c
  Instantiate with
  a = (->)
  t = b->c = (->) b c
  ```

Functions are arrows (3)

- `f >>> g = \a -> g (f a) or`
- `f >>> g = g . f or even`
- `(>>>) = flip (.)`
- `first f = \(b,d) -> (f b,d)`
**Functions are arrows (4)**

Arrow instance declaration for functions:

```haskell
instance Arrow (->) where
  arr     = id
  (>>>)   = flip (.)
  first f = \(b,d) -> (f b, d)
```

**Some arrow laws**

- \((f >>> g) >>> h\) = \(f >>> (g >>> h)\)
- \(arr (f >>> g) = arr f >>> arr g\)
- \(arr id >>> f = f\)
- \(f = f >>> arr id\)
- \(first (arr f) = arr (first f)\)
- \(first (f >>> g) = first f >>> first g\)

**The loop combinator (1)**

Another important operator is `loop`: a fixed-point operator used to express recursive arrows or feedback:

![Loop diagram](image)

**The loop combinator (2)**

Not all arrow instances support `loop`. It is thus a method of a separate class:

```haskell
class Arrow a => ArrowLoop a where
  loop :: a (b, d) (c, d) -> a b c
```

Remarkably, the four combinators `arr`, `>>,`, `first`, and `loop` are sufficient to express any conceivable wiring!
Some more arrow combinators (1)

second :: Arrow a => a b c -> a (d,b) (d,c)

(*** ) :: Arrow a => a b c -> a d e -> a (b,d) (c,e)

(&&&) :: Arrow a => a b c -> a b d -> a b (c,d)

Some more arrow combinators (2)

As diagrams:

second f

f *** g

f &&& g

Exercise 6

Describe the following circuit using arrow combinators:

a1, a2, a3 :: A Double Double

second f = arr swap >>> first f >>> arr swap
swap (x,y) = (y,x)

(*** ) :: Arrow a => a b c -> a d e -> a (b,d) (c,e)
f *** g = first f >>> second g

(&&&) :: Arrow a => a b c -> a b d -> a b (c,d)
f &&& g = arr (
x -> (x,x)) >>> (f *** g)
Exercise 3: Describe the following circuit using arrow combinators:

\[
\text{circuit}_v1 :: \text{A Double Double} \\
\text{circuit}_v1 = (\text{a1} &&& \text{arr id}) \\
\quad \gg (\text{a2} *** \text{a3}) \\
\quad \gg \text{arr (uncurry (+))}
\]

Exercise 3: Describe the following circuit:

\[
\text{circuit}_v2 :: \text{A Double Double} \\
\text{circuit}_v2 = \text{arr } (\lambda x \to (x,x)) \\
\quad \gg \text{first a1} \\
\quad \gg (\text{a2} *** \text{a3}) \\
\quad \gg \text{arr (uncurry (+))}
\]

Exercise 6: Another solution

\[
\text{circuit}_v2 :: \text{A Double Double} \\
\text{circuit}_v2 = \text{arr } (\lambda x \to (x,x)) \\
\quad \gg \text{first a1} \\
\quad \gg (\text{a2} *** \text{a3}) \\
\quad \gg \text{arr (uncurry (+))}
\]

The arrow do notation (1)

Ross Paterson’s do-notation for arrows supports *pointed* arrow programming. Only *syntactic* sugar.

\[
\text{proc } \text{pat} \to \text{do } \{ \text{rec} \}
\]

\[
\text{pat}_1 \leftarrow \text{sfexp}_1 \leftarrow \text{exp}_1 \\
\text{pat}_2 \leftarrow \text{sfexp}_2 \leftarrow \text{exp}_2 \\
\text{...} \\
\text{pat}_n \leftarrow \text{sfexp}_n \leftarrow \text{exp}_n \\
\text{returnA} \leftarrow \text{exp}
\]

Also: let \( \text{pat} = \text{exp} \equiv \text{pat} \leftarrow \text{arr id} \leftarrow \text{exp} \)

The arrow do notation (2)

Let us redo exercise 3 using this notation:

\[
\text{circuit}_v4 :: \text{A Double Double} \\
\text{circuit}_v4 = \text{proc } x \to \text{do} \\
\quad y_1 \leftarrow \text{a1} \leftarrow x \\
\quad y_2 \leftarrow \text{a2} \leftarrow y_1 \\
\quad y_3 \leftarrow \text{a3} \leftarrow x \\
\text{returnA} \leftarrow y_2 + y_3
\]
We can also mix and match:

```
circuit_v5 :: A Double Double
circuit_v5 = proc x -> do
  y2 <- a2 <<< a1 -< x
  y3 <- a3 -< x
  returnA -< y2 + y3
```

Exercise 5: Describe this using only the arrow combinators.

Recursive networks: do-notation:

```
a1, a2 :: A Double Double
a3 :: A (Double,Double) Double
```

Arrows generalize monads: for every monad type there is an arrow, the *Kleisli category* for the monad:

```
newtype Kleisli m a b = K (a -> m b)
instance Monad m => Arrow (Kleisli m) where
  arr f = K (\b -> return (f b))
  K f >>> K g = K (\b -> f b >>= g)
```
Arrows and Monads (2)

But not every arrow is a monad. However, arrows that support an additional apply operation are effectively monads:

\[
\text{apply} :: \text{Arrow } a \Rightarrow a (a \ b \ c, b) c
\]

Exercise 7: Verify that

\[
\text{newtype } M b = M (A () b)
\]

is a monad if \( A \) is an arrow supporting apply; i.e., define return and bind in terms of the arrow operations (and verify that the monad laws hold).

An application: FRP

Functional Reactive Programming (FRP):

- Paradigm for reactive programming in a functional setting:
  - Input arrives incrementally while system is running.
  - Output is generated in response to input in an interleaved and timely fashion.
- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).
- Has evolved in a number of directions and into different concrete implementations.

Yampa

Yampa:

- The most recent Yale FRP implementation.
- Embedding in Haskell (a Haskell library).
- Arrows used as the basic structuring framework.
- Continuous time.
- Discrete-time signals modelled by continuous-time signals and an option type.
- Advanced switching constructs allows for highly dynamic system structure.

Related languages

FRP related to:

- Synchronous languages, like Esterel, Lucid Synchrone.
- Modeling languages, like Simulink.

Distinguishing features of FRP:

- First class reactive components.
- Allows highly dynamic system structure.
- Supports hybrid (mixed continuous and discrete) systems.
FRP applications

Some domains where FRP has been used:
- Graphical Animation (Fran: Elliott, Hudak)
- Robotics (Frob: Peterson, Hager, Hudak, Elliott, Pembeci, Nilsson)
- Vision (FVision: Peterson, Hudak, Reid, Hager)
- GUIs (Fruit: Courtney)
- Hybrid modeling (Nilsson, Hudak, Peterson)

Yampa?

Yampa is a river with long calmly flowing sections and abrupt whitewater transitions in between.

Signal functions

Key concept: functions on signals.

Intuition:
- Signal $\alpha \approx \text{Time} \rightarrow \alpha$
- $x :: \text{Signal } T1$
- $y :: \text{Signal } T2$
- SF $\alpha \beta \approx \text{Signal } \alpha \rightarrow \text{Signal } \beta$
- $f :: \text{SF } T1 T2$

Additionally: causality requirement.

Signal functions and state

Alternative view:

Signal functions can encapsulate state.

$\text{state}(t)$ summarizes input history $x(t')$, $t' \in [0, t]$.

Functions on signals are either:
- **Stateful**: $y(t)$ depends on $x(t)$ and $\text{state}(t)$
- **Stateless**: $y(t)$ depends only on $x(t)$
Yampa and Arrows

SF is an arrow. Signal function instances of core combinators:

• \( \text{arr} :: (a \to b) \to SF ~ a ~ b \)
• \( \text{>>>} :: SF ~ a ~ b \to SF ~ b ~ c \to SF ~ a ~ c \)
• \( \text{first} :: SF ~ a ~ b \to SF ~ (a,c) ~ (b,c) \)
• \( \text{loop} :: SF ~ (a,c) ~ (b,c) \to SF ~ a ~ b \)

But `apply` has no useful meaning. Hence SF is not a monad.

Some further basic signal functions

• \( \text{identity} :: SF ~ a ~ a \)
  \[ \text{identity} = \text{arr} ~ \text{id} \]
• \( \text{constant} :: b \to SF ~ a ~ b \)
  \[ \text{constant} ~ b = \text{arr} ~ (\text{const} ~ b) \]
• \( \text{integral} :: \text{VectorSpace a} \to SF ~ a ~ a \)
• \( \text{time} :: SF ~ a ~ \text{Time} \)
  \[ \text{time} = \text{constant} ~ 1.0 \to \text{integral} \]
• \( (\text{^<<}) :: (b \to c) \to SF ~ a ~ b \to SF ~ a ~ c \)
  \[ f (\text{^<<}) ~ \text{sf} = \text{sf} \to \text{arr} ~ f \]

Example: A bouncing ball

\[ y = y_0 + \int v \, dt \]
\[ v = v_0 + \int -9.81 \]

On impact:
\[ v = -v(t-) \]
(fully elastic collision)

Part of a model of the bouncing ball

Free-falling ball:

```haskell```
type Pos = Double

type Vel = Double

fallingBall ::
  Pos \to Vel \to SF () (Pos, Vel)
fallingBall y0 v0 = proc () \-> do
  v <- (v0 +) \^<< integral \<- -9.81
  y <- (y0 +) \^<< integral \<- v
  returnA \<- (y, v)
```
**Dynamic system structure**

*Switching* allows the structure of the system to evolve over time:

![Diagram](image)

**Example: Space Invaders**

![Space Invaders](image)

**Overall game structure**

![Diagram](image)

**Reading (1)**


Reading (2)


Reading (3)


Reading (4)