This Lecture

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

Monads in Haskell

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

class Monad m where
  -- return :: a -> Maybe a
  return = Just

  -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  Unique b Nothing >>= _ = Nothing
  (Just x) >>= f = f x

This Lecture

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

The Maybe Monad in Haskell

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

  -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  Unique 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:

newtype 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

instance Monad S where
  return a = S (
    
  m >>= f = S <$> 
    
  let (a, s') = unS m s 
    in unS (f a) s'

The do-notiation (1)

Haskell provides convenient syntax for programming with monads:

```
do
  a <- expr1
  b <- expr2
  return expr3
```

is syntactic sugar for

```
expr1 >>= \a ->
expr2 >>= \b ->
return expr3
```
The do-notation (2)
Computations can be done solely for effect, ignoring the computed value:
\[
\begin{align*}
d & \{ c rp_1 \\
c & \{ c rp_2 \\
& \text{return } c rp_3
\end{align*}
\]
is syntactic sugar for
\[
\begin{align*}
crp_1 & \triangleright= \_ \rightarrow \\
crp_2 & \triangleright= \_ \rightarrow \\
& \text{return } c rp_3
\end{align*}
\]

The do-notation (3)
A let-construct is also provided:
\[
\begin{align*}
do & \{ \text{let } a = c rp_1 \\
b & = c rp_2 \\
& \text{return } c rp_3
\end{align*}
\]
is equivalent to
\[
\begin{align*}
do & \{ a \leftarrow \text{return } c rp_1 \\
b & \leftarrow \text{return } c rp_2 \\
& \text{return } c rp_3
\end{align*}
\]

Numbering Trees in do-notation
\[
\begin{align*}
\text{numberTree} & :: \text{Tree } a \rightarrow \text{Tree } \text{Int} \\
\text{numberTree} t & = \text{runS } (\text{ntAux } t) \\
\text{where} & \\
\text{ntAux} & :: \text{Tree } a \rightarrow S (\text{ntAux } t) \\
\text{ntAux} (\text{Leaf } \_) & = \text{do} \\
& \ n \leftarrow \text{get} \\
& \ \text{set } (n + 1) \\
& \ \text{return } \text{Leaf } n \\
\text{ntAux} (\text{Node } t1 \ t2) & = \text{do} \\
& \ t1' \leftarrow \text{ntAux } t1 \\
& \ t2' \leftarrow \text{ntAux } t2 \\
& \ \text{return } \text{Node } t1' \ t2'
\end{align*}
\]

The Compiler Fragment Revisited (1)
Given a suitable “Diagnostics” monad \(D\) that collects error messages, \(\text{enterVar}\) can be turned from this:
\[
\begin{align*}
\text{enterVar} & :: \text{Id } \rightarrow \text{Int } \rightarrow \text{Type } \rightarrow \text{Env} \\
& \rightarrow \text{Either Env ErrorMsgs}
\end{align*}
\]
into this:
\[
\begin{align*}
\text{enterVarD} & :: \text{Id } \rightarrow \text{Int } \rightarrow \text{Type } \rightarrow \text{Env} \\
& \rightarrow D \ \text{Env}
\end{align*}
\]
and then \(\text{identDefs}\) from this ...

The Compiler Fragment Revisited (2)
\[
\begin{align*}
\text{identDefs} \ l \ \text{env} \ [\] & = ([], \text{env}, []) \\
\text{identDefs} \ l \ \text{env} \ ((i,t,e) : ds) & = \\
& ((i,t,e') : ds'', \text{env''}) \\
\text{where} & \\
& e' = \text{identAux } l \ \text{env e} \\
& \text{env'} = \text{enterVar } l \ i \ t \ \text{env} \\
& (ds'', \text{env''}) = \text{identDefs} \ l \ \text{env''} \ ds
\end{align*}
\]
The monadic version is very close to ideal, without sacrificing functionality, clarity, or pureness!

The List Monad
Computation with many possible results, “nondeterminism”:
\[
\begin{align*}
\text{instance Monad } [] \text{ where} \\
& \text{return } a = [a] \\
& m >>= f = \text{concat } (\text{map } f \ m) \\
& \text{fail } s = []
\end{align*}
\]
Example:
\[
\begin{align*}
x & \leftarrow [1, 2] \\
y & \leftarrow ['a', 'b']
\end{align*}
\]

The Reader Monad
Computation in an environment:
\[
\begin{align*}
\text{instance Monad } ((\rightarrow) \ e) \text{ where} \\
& \text{return } a = \text{const } a \\
& m >>= f = \text{\_e} \rightarrow f \ (m \ e) \\
\text{getEnv} & :: ((\rightarrow) \ e) \ e \\
\text{getEnv} & = \text{id}
\end{align*}
\]

(Suffix D just to remind us the types have changed.)
The Haskell IO Monad

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

```haskell
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:

```haskell
newtype SE s a = SE (s -> Maybe (a, s))
```

However:

- Not always obvious how: e.g., should the combination of state and error have been.
- Duplication of effort: similar patterns related to specific effects are going to be repeated over and over in the various combinations.

Monad Transformers (2)

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:

```haskell
T :: ( * -> * ) -> ( * -> * )
```

Additionally, a monad transformer **adds** computational effects. A mapping `lift` from computations in the underlying monad to computations in the transformed monad is needed:

```haskell
lift :: M a -> T M a
```

Monad Transformers in Haskell (2)

These requirements are captured by the following (multi-parameter) type class:

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

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

Classes for Specific Effects

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

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

```haskell
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:

```haskell
newtype I a = I aunI (I a) = a
```

instance Monad I where

```haskell
return a = I a
m >>= f = f (unI m)
runI :: I a -> arunI = unI
```

The Error Monad Transformer (1)

```haskell
newtype ET m a = ET (m (Maybe a))
```

Any monad transformed by `ET` is a monad:

```haskell
instance Monad ET m a 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)
```

```haskell
unET (ET m) = m
```
The Error Monad Transformer (2)

We need the ability to run transformed monads:

\[
\text{runET} :: \text{Monad } m \Rightarrow \text{ET } m \ a \rightarrow m \ a
\]

\[
\text{runET} \ etm = \begin{cases} ma & \text{if } ma \neq \text{Nothing} \\ \text{error } \text{"Should not happen"} & \text{otherwise} \end{cases}
\]

ET is a monad transformer:

\[\text{instance Monad } m \Rightarrow \text{MonadTransformer } \text{ET } m \text{ where} \]

\[\text{lift } m = \text{ET } (\text{m } \gg \gg \ \lambda a \to \text{return (Just a)})\]

The Error Monad Transformer (3)

Any monad transformed by ET is an instance of E:

\[\text{instance Monad } m \Rightarrow \text{E } \text{(ET } m) \text{ where} \]

\[\text{eFail } \text{= ET } (\text{return Nothing}) \]

\[\text{m } \text{\"eHandle\" } m2 = \text{ET } \text{\{ do } ma \leftarrow \text{unET } m1 \\
\text{case } ma \text{ of} \text{ Just } a \rightarrow \text{return a} \\ \text{Nothing} \rightarrow \text{error } \text{\"Should not happen\"} \text{\})} \]

The Error Monad Transformer (4)

A state monad transformed by ET is a state monad:

\[\text{instance S } m \ s \Rightarrow \text{S } \text{(ET } m \ s) \text{ where} \]

\[\text{sSet } s \to \text{ lift } (\text{sSet } s) \]

\[\text{sGet } = \text{lift } \text{sGet} \]

Exercise 2: Running Transf. Monads

Let

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

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.

Exercise 2: Solution

\[\text{ex2 } :: \text{ET I Int} \]

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

\[\text{ex2result } :: \text{Int} \]

\[\text{ex2result } = \text{runI} \ (\text{runET } \text{ex2})\]

Exercise 3: Effect Ordering

Consider the code fragment

\[\text{ex3a } :: \text{(ST Int (ET I)) Int} \]

\[\text{ex3a } = (\text{sSet } 42 \gg \text{eFail}) \text{\"eHandle\" } \text{sGet}\]

Note that the exact same code fragment also can be typed as follows:

\[\text{ex3b } :: \text{(ET (ST Int I)) Int} \]

\[\text{ex3b } = (\text{sSet } 42 \gg \text{eFail}) \text{\"eHandle\" } \text{sGet}\]

What is

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

\[\text{runI } (\text{runST } (\text{runET } \text{ex3b } 0))\]
**Exercise 3: Solution**

\[
\begin{align*}
\text{runI (runET (runST ex3a 0))} &= 0 \\
\text{runI (runST (runET ex3b) 0)} &= 42
\end{align*}
\]

Why? Because:

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

**Exercise 4: Alternative ST?**

To think about.

Could \text{ST} have been defined in some other way, e.g.

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

or perhaps

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

**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:

A combinator can be defined that captures this idea:

\[
(\ggg) :: B \: a \: b \to B \: b \: c \to B \: a \: c
\]

**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’ \text{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 \text{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 -> c) -> a b c
  - composition: (\ggg) :: a b c -> a c d -> a b d
  - widening: \text{first} :: a b c -> a (b, d) (c, d)
- A set of algebraic laws that must hold.

**What is an arrow? (2)**

These diagrams convey the general idea:

**The Arrow class**

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

\[
\begin{align*}
\text{class Arrow a where} \\
\text{arr} &:: (b \to c) \to a \: b \: c \\
\text{ggg} &:: a \: b \: c \to a \: c \: d \to a \: b \: d \\
\text{first} &:: a \: b \: c \to a \: (b, d) \: (c, d)
\end{align*}
\]
Functions are arrows (1)

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

**Exercise 5:** Suggest suitable definitions of
- `arr`
- `>>>(`) for this case!

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

Functions are arrows (2)

**Solution:**
- \( arr = id \)
- `first` ar\(\text{r} \):
  \[
  \text{id} :: t \rightarrow t
  \]
  \[
  arr :: (b\rightarrow c) \rightarrow a \ b \ c
  \]

Instantiate with:
- \( a = \rightarrow \)
- \( t = b \rightarrow c = (\rightarrow) b \ c \)

Functions are arrows (3)

- \( f >>> g = \lambda a \rightarrow g (f \ a) \) **or**
- \( f >>> g = g \ . \ f \) **or even**
- `>>>(`) = `flip (.)`
- `first f = \( \backslash (b,d) \rightarrow (f \ b,d) \)

Functions are arrows (4)

**Arrow instance declaration for functions:**

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

Some arrow laws

- \( (f >>> g) >>> h = f >>> (g >>> h) \)
- `arr (f >>> g) = arr f >>> arr g`
- \( f >>> arr id = f \)
- \( \text{first} (arr f) = arr (\text{first} f) \)
- \( \text{first} (f >>> g) = \text{first} f >>> \text{first} g \)

The loop combinator (1)

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

![loop f diagram]

Some more arrow combinators (1)

- \( \text{second} :: \Arrow a \Rightarrow a \ b \ c \rightarrow a \ (d,b) \ (d,c) \)
- \( (*** :: \Arrow a \Rightarrow a \ (b,d) \ (c,e) \rightarrow a \ b \ c \)
- \( (&&& :: \Arrow a \Rightarrow a \ b \ d \rightarrow a \ b \ (c,d) \)

Some more arrow combinators (2)

As diagrams:

![second f diagram]

![loop f diagram]

![f *** g diagram]
Some more arrow combinators (3)

second :: Arrow a => a b c -> a (d,b) (d,c)
second f = arr swap >>> first f >>> arr swap

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

Exercise 6

Describe the following circuit using arrow combinators:

```
 a1         a2
  |         |
  v         v
a3
```

Exercise 6: One solution

```
circuit_v1 :: A Double Double
circuit_v1 = (a1 (***) arr id)
            >>> (a2 *** a3)
            >>> arr (uncurry (+))
```

Exercise 3: Describe the following circuit:

```
 a1         a2
  |         |
  v         v
a3
```

Exercise 3: Describe the following circuit using arrow combinators:

```
circuit_v4 :: A Double Double
circuit_v4 = proc x -> do
            y1 <- a1 <<< a1 <<< x
            y2 <- a2 <<< y1
            y3 <- a3 <<< (x, y)
            returnA <- y2 + y3
```

Exercise 5: Describe this using only the arrow combinators.

The arrow do notation (1)

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

```
proc pat -> do [ rec ]
    pat1 <- sfexp1 <<< exp1
    pat2 <- sfexp2 <<< exp2
    ...
    patn <- sfexpn <<< expn
    returnA <- exp
```

Also: `let pat = exp` is **equivalent** to `pat <- arr id <<< exp`

The arrow do notation (2)

Let us redo exercise 3 using this notation:

```
circuit_v4 :: A Double Double
circuit_v4 = proc x -> do
            y1 <- a1 <<< a1 <<< x
            y2 <- a2 <<< y1
            y3 <- a3 <<< (x, y)
            returnA <- y2 + y3
```

The arrow do notation (4)

Recursive networks: `do`-notation:

```
 a1         a2
  |         |
  v         v
a3
```

Exercise 5: Describe this using only the arrow combinators.

The arrow do notation (5)

```
circuit <- proc x -> do
    rec
    y1 <- a1 <<< x
    y2 <- a2 <<< y1
    y3 <- a3 <<< (x, y)
    let y = y2 + y3
    returnA <- y
```
Arrows generalize monads: for every monad type there is an arrow, the *Kleisli category* for the monad:

```haskell
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)
```

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

```haskell
apply :: Arrow a => a (a b c, b) c
```

Exercise 7: Verify that

```haskell
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.

**FRP related to:**

- Synchronous languages, like Esterel, Lucid Synchron.
- 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)

**Related languages**

FRP related to:

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

**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.

**Yampa?**

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

A good metaphor for hybrid systems!

**Signal functions**

Key concept: *functions on signals*.

![Diagram of signal functions](image)

Intuition:

\[ \text{Signal } \alpha \approx \text{Time} \to \alpha \]

\[ x :: \text{Signal } T1 \]

\[ y :: \text{Signal } T2 \]

\[ \text{SF } \alpha \beta \approx \text{Signal } \alpha \to \text{Signal } \beta \]

\[ f :: \text{SF } T1 T2 \]

Additionally: *causality* requirement.

**Signal functions and state**

Alternative view:

Signal functions can encapsulate *state*.

![Diagram of signal functions and state](image)

\[ \text{state}(t) \] 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:

- `arr :: (a -> b) -> SF a b`
- `>>> :: SF a b -> SF b c -> SF a c`
- `first :: SF a b -> SF (a,c) (b,c)`
- `loop :: SF (a,c) (b,c) -> SF a b`

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

Some further basic signal functions

- `identity :: SF a aidentity = arr id`
- `constant :: b -> SF a bconstant b = arr (const b)`
- `integral :: VectorSpace a =>SF a a`
- `time :: SF a Time`
  `time = constant 1.0 >>> integral`
- `(^<<) :: (b->c) -> SF a b -> SF a cf (^<<) sf = sf >>> arr f`

Example: A bouncing ball

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

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

Free-falling ball:

```
type Pos = Double
type Vel = Double

fallingBall :: Pos -> Vel -> 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:

Example: Space Invaders

```
\begin{align*}
\text{bullet} &\rightarrow \text{alien} \\
\text{alien} &\rightarrow \text{killOrSpawn} \\
\text{gun} &\rightarrow \text{ObjInput} \\
\text{ObjOutput} &\rightarrow \text{ObjInput}
\end{align*}
```

Reading (1)


Reading (2)

Reading (3)


Reading (4)