This Lecture

- Monads in Haskell
- Some standard monads
- Combining effects: monad transformers

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.

The Maybe Monad in Haskell

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

  -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  (Just x) >>= f = f x
```
Exercise 1: A State Monad in Haskell

Haskell 2010 does not permit type synonyms to be instances of classes. Hence we have to define a new type:

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

```haskell
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:

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

is syntactic sugar for

```haskell
exp1 >>= \a ->
ex2 >>= \b ->
return exp3
```

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 >>= \_ ->
ex2 >>= \_ ->
return exp3
```

The do-notation (3)

A let-construct is also provided:

```haskell
do
  let a = exp1
      b = exp2
  return exp3
```

is equivalent to

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

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 <- getset (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, `enterVar` can be turned from this:

$$\text{enterVar} :: \text{Id} \rightarrow \text{Int} \rightarrow \text{Type} \rightarrow \text{Env}$$

into this:

$$\text{enterVarD} :: \text{Id} \rightarrow \text{Int} \rightarrow \text{Type} \rightarrow \text{Env}$$

($\text{Suffix “D” just to remind us the types have changed.}$)

And then `identDefs` from

$$\text{identDefs} :: \text{Int} \rightarrow \text{Env} \rightarrow \{(\text{Id}, \text{Type}, \text{Exp}())\} \rightarrow \{(\text{Id}, \text{Type}, \text{Exp Attr}), \text{Env}, \text{[ErrorMsg]}\}$$

into

$$\text{identDefsD} :: \text{Int} \rightarrow \text{Env} \rightarrow \{(\text{Id}, \text{Type}, \text{Exp}())\}$$

with the function definition changing from ...
The Compiler Fragment Revisited (4)

Compare with the “core” identified earlier!

identDefs l env [] = ([], env)
identDefs l env ((i,t,e) : ds) = ((i,t,e’) : ds’, env’’)
where
  e’ = identAux l env e
  env’ = enterVar i l t env
  (ds’, env’’) = identDefs l env’ ds

The monadic version is very close to this “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
  getEnv :: ((->) 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 :: (* -> *) -> (* -> *)
  ```
- Additionally, a monad transformer **adds** computational effects. A mapping `lift` from computations in the underlying monad to computations in the transformed monad is needed:
  
  ```
  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, Monad (t m)) => MonadTransformer t m where
  lift :: m a -> 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:

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

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 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)

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

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:

```haskell
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:**

```haskell
instance Monad m => MonadTransformer ET m where
  lift m = ET (m >>= \a -> return (Just a))
```

The Error Monad Transformer (3)

Any monad transformed by `ET` is an instance of `E`:

```haskell
instance Monad m => E (ET m) where
  eFail = ET (return Nothing)
  m1 `eHandle` m2 = ET $ do
    ma <- unET m1
    case ma of
      Nothing -> unET m2
      Just _ -> return ma
```

The Error Monad Transformer (4)

A state monad transformed by `ET` is a state monad:

```haskell
instance S m s => S (ET m) s where
  sSet s = lift (sSet s)
  sGet = lift sGet
```

Exercise 2: Running Transf. Monads

Let

```haskell
ex2 = eFail `eHandle` return 1
```

1. Suggest a possible type for `ex2`. (Assume `1 :: Int`.)
2. Given your type, use the appropriate combination of “run functions” to run `ex2`. 
Exercise 2: Solution

```haskell
ex2 :: ET I Int
ex2 = eFail `eHandle` return 1

ex2result :: Int
ex2result = runI (runET ex2)
```

The State Monad Transformer (1)

```haskell
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 \rightarrow return (a, s))
m >>= f = ST ($
\s \rightarrow \do
(a, s`) <- unST m s
unST (f a) s`
```

The State Monad Transformer (2)

We need the ability to run transformed monads:

```haskell
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:

```haskell
instance Monad m => MonadTransformer (ST s) m where
lift m = ST ($
\s \rightarrow m \gg= \a \rightarrow
\return (a, s))
```

The State Monad Transformer (3)

Any monad transformed by ST is an instance of S:

```haskell
instance Monad m => S (ST s m) s where
sSet s = ST ($
_; s \rightarrow return ())
sGet = ST ($
\s \rightarrow return (s, s))
```

An error monad transformed by ST is an error monad:

```haskell
instance E m => E (ST s m) where
eFail = lift eFail
m1 `eHandle` m2 = ST $ $
\s \rightarrow
unST m1 s `eHandle` unST m2 s
```
Exercise 3: Effect Ordering

Consider the code fragment

\[
\text{ex3a :: (ST Int (ET I)) Int} \\
\text{ex3a = (sSet 42 >> eFail) 'eHandle' sGet}
\]

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

\[
\text{ex3b :: (ET (ST Int I)) Int} \\
\text{ex3b = (sSet 42 >> eFail) 'eHandle' sGet}
\]

What is

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

Exercise 3: Solution (1)

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

Why? Because:

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

Exercise 3: Solution (2)

Note that

\[
\text{ET (ST s I) a} \equiv s \rightarrow (\text{Maybe a, s})
\]

results in a notion of a \textit{shared, global} state, while

\[
\text{ST s (ET I) a} \equiv s \rightarrow \text{Maybe (a, s)}
\]

has a \textit{transactional} flavour: only if a computation succeeds will any effects from that computation be taken into account.

\textit{Both} are natural and useful; hence there is no “right” or “wrong” ordering.

Exercise 4: Alternative ST?

To think about.

Could \textit{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.

Reading (1)


Reading (2)


• Mauro Jaskelioff. Modular Monad Transformers. In European Symposium on Programming (ESOP'09), 2009.