Monads

COMP2209 - Programming III

Dr Julian Rathke
Safe Division using Maybe

• Let’s consider an example of error handling using the Maybe Applicative

• We’ll use the data type (we could add other arithmetic but this is sufficient for the example)

```haskell
data Expr = Val Int | Div Expr Expr
```

• A naive interpreter function for this language might be

```haskell
eval :: Expr -> Int
eval (Val n) = n
eval (Div x y) = eval x `div` eval y
```

• But of course this falls over when trying to divide by 0
Safe Division using Maybe

- Let’s rewrite the evaluation function using Maybe to handle the error case

```haskell
eval :: Expr → Maybe Int
eval (Val n) = Just n
eval (Div x y) = case eval x of
    Nothing → Nothing
    Just n → case eval y of
        Nothing → Nothing
        Just m → safediv n m

safediv :: Int → Int → Maybe Int
safediv _ 0 = Nothing
safediv n m = Just (n `div` m)
```

Each time we call `eval` we may get `Nothing` as a return value
Safe Division using Maybe

• We’re ever so smart now though and see that the standard error handling cases in this long-winded definition of eval could be hidden in the structured application <*> for Maybe.

```
 eval :: Expr -> Maybe Int
 eval (Val n) = pure n
 eval (Div x y) = pure safediv <*> eval x <*> eval y
```

Nice try but no banana!

• Ah, see the type of safediv :: Int -> Int -> Maybe Int is not what is needed here. To use it with pure as above it should have type Int -> Int -> Int

• Or to write a bespoke version of pure safediv we would need type Maybe (Int -> Int -> Int)

• Neither of these are right. These types do not allow for the case where the error is produced in actually evaluating safediv
A different pattern

• We can only conclude that this eval function does not fit the general pattern of structured application.

• We’ve seen that Functors \( \text{fmap} :: (a \to b) \to f~a \to f~b \) take a function and apply that function to a structured value to return a structured result.

• We’ve seen that Applicatives \( \text{<*>} :: f~(a \to b) \to f~a \to f~b \) generalise this to take a structured function and a structured value to return a structured result.

• We now need something that can take a structured value and a function that may produce a structured result and apply the latter to the former to produce a structured result.

\[ \text{>>=} :: f~a \to (a \to f~b) \to f~b \]

This operation is called \textbf{bind}

Confusingly, the function and argument have swapped around in the type here.

This will take some digesting.
Rewriting eval using >>=

- Let’s see how this new operator can help tidy up the eval code
- For the Maybe type we have

\[
\text{>>=} :: \text{Maybe } a \rightarrow (a \rightarrow \text{Maybe } b) \rightarrow \text{Maybe } b
\]

\[
\text{mx } \text{>>=} f = \text{case mx of}
\]

\[
\begin{align*}
& \text{Nothing } \rightarrow \text{Nothing} \\
& \text{Just } x \rightarrow f\ x
\end{align*}
\]

\[
\text{eval} :: \text{Expr } \rightarrow \text{Maybe } \text{Int}
\]

\[
\begin{align*}
\text{eval} (\text{Val } n) &= \text{Just } n \\
\text{eval} (\text{Div } x \ y) &= \text{eval} x \text{>>=} \n \rightarrow \\
& \quad \text{eval} y \text{>>=} \m \rightarrow \\
& \quad \text{safediv } n \ m
\end{align*}
\]

This is just the standard error handling code we used before!

All the standard Maybe error handling is hidden in >>=

Compare this with the explicit version on Slide 4!
A typical usage of >>=

• The typical way in which bind is used is exactly as we did in eval above.

• Some structured computation \( m_i \) is evaluated, the result is unpacked and bound to \( x_i \) which can then be used for the final return value.

• This should look a bit familiar - in fact, Haskell has a syntactic sugar for expressions of this exact form.

• This is exactly the same notation used for IO actions in Haskell!

• Because it IO actions are an instance of this general pattern in fact.
The simplest version of eval

- We can use the “do” notation to rewrite our increasingly stretched example function eval

```haskell
eval :: Expr -> Maybe Int

eval (Val n) = Just n

eval (Div x y) = do n <- eval x
                   m <- eval y
                   safediv n m
```

- So when are we allowed to use this “do” notation?
  - For any type operator \( f \) \( a \) where we have defined the bind operator.
  - How do we know if there is a bind operator defined?
  - There is a class for that - the **Monad** class
Class Monad

class Applicative m ⇒ Monad m where
    return :: a → m a
    (>>=) :: m a → (a → m b) → m b
    return = pure

Every Monad is an Applicative is a Functor

Default definition - return in the class only due to legacy from before Applicatives were introduced

Any instance of Monad supports “do” notation

All of the following are instances of Monad:

- Maybe a
  - Calculate values with error handling
- IO a
  - Calculate values with IO side-effects
- [ a ]
  - Calculate values with multiple answers
Functors, Applicatives, Monads - a visual aid

**Functors**: \( g \) applies functions to structured values and produces structured values. \( g \) applies \( \circ \) to \( \bullet \) to produce \( \bigcirc \).

**Applicatives**: \( \langle\ast\rangle \) applies structured functions to structured values. \( g \) applies \( \circ \) to \( \bullet \) to produce \( \bigcirc \).

**Monads**: \( >>= \) applies functions that produce structured values to structured values. \( g \) applies \( \circ \) to \( \bullet \) to produce \( \bigcirc \).
Examples Time

• Monads seem very abstract and theoretical but are they actually any use?

• Yes! You have been using them to do IO with all of its error handling without even realising it.

• They provide a uniform mechanism for dealing with the extra bits of wiring needed to sequence together computations that have extra complications (such as errors, IO, non-determinism).

• Monads expose the fact that these extra bits of wiring are instances of the same programming idiom

• Examples are the most convincing way of seeing this so let’s look at some now.

• We’ve already seen an example using the Maybe monad, and loads of examples using IO. So let’s look at Lists.
The List Monad

- The implementation of bind for lists is straightforward:

```haskell
instance Monad [] where
  (>>=) :: [a] → (a → [b]) → [b]
xs >>= f = [ y | x ← xs, y ← f x ]
```

- Apply f to every element in xs and gather together the multiple results each time as a list.
- This could be defined as `concat (map f xs)` instead.
- For example

```
> [3,4,5] >>= \x → [x,-x]
[3,-3,4,-4,5,-5]
```

Note that `[] >>= f` is always `[]` no matter what f is.
This is analogous to Nothing >>= f always being Nothing for the Maybe monad.
A Knights Tour

- Imagine moving a Knight on a standard chess board.
- They can move (in any orientation), two squares forward and one square left/right.
- Given any starting position calculate all possible squares that a knight can reach in 3 moves.
- We’ll represent a position as a pair of Int

```haskell
type KnightPos = (Int, Int)
```

- We can code the legal moves from a given position easily

```haskell
moveKnight :: KnightPos \rightarrow [ KnightPos ]
moveKnight (c,r) = filter onBoard
    [ (c+2,r-1), (c+2,r+1), (c-2,r-1), (c-2,r+1),
      (c+1,r-2), (c+1,r+2), (c-1,r-2), (c-1,r+2) ]
where onBoard (c,r) = c `elem` [1..8] && r `elem` [1..8]
```
A Knights Tour

• We now just need to sequence the moveKnight function 3 times from the given starting position.
• This is where the List monad kicks in

\[
\text{in3moves} :: \text{KnightPos} \rightarrow [\text{KnightPos}]
\]
\[
\text{in3moves \ start = return \ start >>= \ moveKnight >>= \ moveKnight >>= \ moveKnight}
\]

return just makes a singleton list

each move creates a list of squares reachable from each previously reachable square.

these lists are flattened in to a single result list
The State Monad

• Remember in our Hangman game that we carried around a bit of state in the function?

• We passed the word to be guessed and also the answer so far as a string that recorded which letters had been guessed at each unfolding of the recursion.

• The wordToGuess is unchanged each time but the answerSoFar behaves like mutable state.

• This diagram should look familiar.

• It looks like the description of the IO monad in which the state of the world is passed and modified.

• We can build a general purpose State monad exactly this way.
The State Monad

- We first define a type operator to represent this state transformation.
- We can’t use a type synonym as these can’t be made into Monads.
- But we can use a newtype declaration with a dummy constructor to achieve much the same thing.
- We need to decide what sort of state we are storing - let’s just store a single Int for this example:

```haskell
newtype ST a = S (State -> (a, State))
```

- It will be useful to define an application function that removes the dummy constructor too.

```haskell
app :: ST a -> State -> (a, State)
ap (S st) = st
```
We need to define ST as a Functor, Applicative and Monad instance

instance Functor ST where
  fmap :: (a -> b) -> ST a -> ST b
  fmap g st = S (\s ->
      let (x,s') = app st s
      in (g x , s'))

instance Applicative ST where
  pure :: a -> ST a
  pure x = S (\s -> (x,s))
  (<*>) :: ST (a -> b) -> ST a -> ST b
  stf <$> stx = S (\s ->
      let (f,s') = app stf s
          (x,s'') = app stx s'
      in (f x , s''))

In each case we just apply the state transformer to the current state to get the new state and return value.
The State Monad

We need to define ST as a Functor, Applicative and Monad instance

```
instance Monad ST where
  (>>=) :: ST a -> (a -> ST b) -> ST b
st >>= f = S ( \s ->
  let (x,s') = app stx s
  in app (f x) s' )
```

This just evaluates the updated state when calculating the argument x to f, it then feeds x to f along with the updated state.
Relabelling a Tree - 3 ways

• Given the data type

```haskell
data LTree a = Leaf a | Node (Tree a) (Tree a)
```

• We will write a function that relabels a given Tree so that each of its nodes are labelled with a fresh integer value.

```haskell
relabel :: Tree a → Tree Int
```

• We could program this directly in Haskell by threading the “next” integer to be used throughout a recursive function.

```haskell
relabel t = fst $ rlabel t 0
rlabel :: Tree a → Int → (Tree Int, Int)
rlabel (Leaf _) n = (Leaf n, n+1)
rlabel (Node l r) n = (Node l' r', n'')
where (l',n') = rlabel l n
      (r',n'') = rlabel r n'
```

This is okay as a solution but can be tidied using the ST monad.

This threading of state is a standard trick.
Relabelling a Tree - 3 ways

- Let’s rewrite this relabelling function in applicative style

```haskell
fresh :: ST Int
fresh = S (\n \rightarrow (n, n+1))
```

A simple State Transformer

```haskell
relabel t = fst $ app (alabel t) 0
alabel :: Tree a \rightarrow ST (Tree Int)
alabel (Leaf _) = Leaf <$> fresh
alabel (Node l r) = Node <$> alabel l <*> alabel r
```

Note that \( g <$> x \) is defined as \((\text{pure } g) <*> x\)

This is so close to a pure recursive definition!
Relabelling a Tree - 3 ways

Finally, let’s write relabel using “do” notation and the ST monad.

```
fresh :: ST Int
fresh = S (\n -> (n,n+1))
```

```
relabel t = fst $ app (mlabel t) 0

mlabel :: Tree a -> ST (Tree Int)
mlabel (Leaf _) = do n <- fresh
                    return (Leaf n)

mlabel (Node l r) = do l' <- mlabel l
                       r' <- mlabel r
                       return (Node l' r')
```

This is also very clear recursive code!

Applicative vs Monads?
It’s a matter of taste in this example.
Exploring Control.Monad

- The package Control.Monad is where the Functor and Monad classes are actually defined.
- Also in this package are a number of useful generic functions that work with any monad instance.
- In particular we see a versions of list map that can be used for functions that return any monad type

\[
\text{mapM} :: \text{Monad } m \Rightarrow (a \to m b) \to [a] \to m [b]
\]
\[
\text{mapM } f \ [x:xs] = \text{do } y \leftarrow f x
\]
\[
\text{ys } \leftarrow \text{mapM } f \ xs
\]
\[
\text{return } (y:ys)
\]
Example of mapM

```
conv :: Char → Maybe Int
conv c | isDigit c = Just (digitToInt c)
        | otherwise = Nothing
```

Converts a character to its integer value whenever it is a digit but returns Nothing otherwise.

```
> mapM conv "1234"
Just [1,2,3,4]

> mapM conv "123a"
Nothing
```

All characters are digits and all are converted

A single non-digit character causes a Nothing return overall due to the error handing in "do" in the Maybe monad.
filterM

- Similar to mapM we have filterM that can filter elements from a list according to a monad type returning predicate

```haskell
filterM :: Monad m => (a → m Bool) → [a] → m [a]
filterM p [] = return []
filterM p (x:xs) = do b <- p x
  ys <- filterM p xs
  return (if b then x:ys else ys)
```

Here is an amazingly concise implementation of the powerset of a list

```
powerlist = filterM (\x → [True, False])
```

> powerlist [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[[]]]
Monad Laws

• Just as with Functors and Applicatives there are equational laws that Monad implementations are expected to satisfy.
• Again, there is no compiler support for this in GHC.

**Monad Law 1:**
\[
\text{return } x \gg= f = f \ x
\]

**Monad Law 2:**
\[
mx \gg= \text{return } = mx
\]

**Monad Law 3:**
\[
(mx \gg= f) \gg= g = mx \gg= (\lambda x \rightarrow (f \ x \gg= g))
\]

These two laws say that return must act like an identity function for \( \gg= \) composition.

This third law says that \( \gg= \) must be associative (upto \( \backslash \) binding to get the types right).
Monads are not just for Haskell …

• Remember that Monads are a design pattern for programming.
• The bind operation >>= is used as a structuring mechanism for hiding “wiring” in various settings.
• Because they are just a design pattern they are not limited to Haskell. You can program in monadic style in many languages.
• Haskell provides first-class support for them at type level and they are used for the Haskell IO model but they can be used without this.
• You can program using Monads in Java! Or in JavaScript.
• In fact, in the next few lectures we’ll look at how to take Functional Programming as a programming style and use it in these imperative languages.
YOUR QUESTIONS

Next Lecture:
Functional Programming in Java