

COMP4075: Lecture 9

Monads in Haskell

Henrik Nilsson

University of Nottingham, UK

This Lecture

- Monads in Haskell
- The Haskell Monad Class Hierarchy
- Some Standard Monads and Library Functions

Monads in Haskell (1)

In Haskell, the notion of a monad is captured by a **Type Class**. In principle (but not quite from GHC 7.8 onwards):

```
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.

COMP4075: Lecture 9 - p.137

COMP4075: Lecture 9 - p.137

COMP4075: Lecture 9 - p.137

Monads in Haskell (2)

The Haskell monad class has two further methods with default definitions:

```
(>>) :: m a -> m b -> m b
m >> k = m >>= \_ -> k
fail :: String -> m a
fail s = error s
```

(However, *fail* will likely be moved into a separate class *MonadFail* in the future.)

COMP4075: Lecture 9 - p.137

COMP4075: Lecture 9 - p.137

COMP4075: Lecture 9 - p.137

The *Maybe* Monad in Haskell

```
instance Monad Maybe where
  return = Just
  Nothing >>= _ = Nothing
  (Just x) >>= f = f x
```

COMP4075: Lecture 9 - p.137

COMP4075: Lecture 9 - p.137

COMP4075: Lecture 9 - p.137

The Monad Type Class Hierachy (1)

Monads are mathematically related to two other notions:

- Functors
- Applicative Functors (or just Applicatives)

Every monad is an applicative functor, and every applicative functor (and thus monad) is a functor.

Class hierarchy:

```
class Functor f where ...
class Functor f => Applicative f where ...
class Applicative m => Monad m where ...
```

COMP4075: Lecture 9 - p.137

COMP4075: Lecture 9 - p.137

COMP4075: Lecture 9 - p.137

The Monad Type Class Hierachy (2)

For example, *fmap* can be defined in terms of *>>=* and *return*, demonstrating that a monad is a functor:

```
fmap f m = m >>= \x -> return (f x)
```

A consequence of this class hierarchy is that to make some *T* an instance of *Monad*, an instance of *T* for both *Functor* and *Applicative* must also be provided.

Note: Not a mathematical necessity, but a result of how these notions are defined in Haskell at present. E.g. monads can be understood in isolation.

COMP4075: Lecture 9 - p.137

COMP4075: Lecture 9 - p.137

COMP4075: Lecture 9 - p.137

Applicative Functors (1)

An applicative functor is a functor with application, providing operations to:

- embed pure expressions (*pure*), and
- sequence computations and combine their results (*<*>*)

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  (*>) :: f a -> f b -> f b
  (<*) :: f a -> f b -> f a
```

COMP4075: Lecture 9 - p.137

COMP4075: Lecture 9 - p.137

COMP4075: Lecture 9 - p.137

Applicative Functors (2)

- Like monads, applicative functors is a notion of computation.
- The key difference is that the result of one computation is not made available to subsequent computations. As a result:
 - The **structure** of a computation is static.
 - **Scope** for running computations in **parallel**.
 - Whether the computations **actually** can be carried in parallel depends on what the specific effects of the applicative in question are.

COMP4075: Lecture 9 - p.137

COMP4075: Lecture 9 - p.137

COMP4075: Lecture 9 - p.137

Applicative Functors (3)

Laws:

$$\begin{aligned} \text{pure } id \langle\&*\rangle v &= v \\ \text{pure } (\circ) \langle\&*\rangle u \langle\&*\rangle v \langle\&*\rangle w &= u \langle\&*\rangle (v \langle\&*\rangle w) \\ \text{pure } f \langle\&*\rangle \text{pure } x &= \text{pure } (f x) \\ u \langle\&*\rangle \text{pure } y &= \text{pure } (\$y) \langle\&*\rangle u \end{aligned}$$

Default definitions:

$$\begin{aligned} u \&* v &= \text{pure } (\text{const } id) \langle\&*\rangle u \langle\&*\rangle v \\ u \&* v &= \text{pure } \text{const } \langle\&*\rangle u \langle\&*\rangle v \end{aligned}$$

COMP4075: Lecture 9 – p.13/37

Instances of Alternative

instance Alternative [] where

$$\begin{aligned} \text{empty} &= [] \\ \langle\langle| \rangle\rangle &= (+) \end{aligned}$$

instance Alternative Maybe where

$$\begin{aligned} \text{empty} &= \text{Nothing} \\ \text{Nothing } \langle\langle| \rangle\rangle r &= r \\ l \langle\langle| \rangle\rangle _ &= l \end{aligned}$$

COMP4075: Lecture 9 – p.13/37

Example: Applicative Parser (3)

```
command :: Parser Command
command =
  pure If
    <* kwd "if"   <*> expr
    <* kwd "then" <*> command
    <* kwd "else" <*> command
  <|> pure Block
    <* kwd "begin"
    <*> many (command <* symb ";" )
    <* kwd "end"
```

COMP4075: Lecture 9 – p.13/37

Instances of Applicative

instance Applicative [] where

$$\begin{aligned} \text{pure } x &= [x] \\ fs \langle\&*\rangle xs &= [f x \mid f \leftarrow fs, x \leftarrow xs] \end{aligned}$$

instance Applicative Maybe where

$$\begin{aligned} \text{pure} &= \text{Just} \\ \text{Just } f \langle\&*\rangle m &= \text{fmap } f m \\ \text{Nothing} \langle\&*\rangle _ &= \text{Nothing} \end{aligned}$$

COMP4075: Lecture 9 – p.13/37

Example: Applicative Parser (1)

Applicative functors are frequently used in the context of parsing combinators. In fact, that is where their origin lies.

A *Parser* computation allows reading of input, fails if input cannot be parsed, and supports trying alternatives:

instance Applicative Parser where...

instance Alternative Parser where...

COMP4075: Lecture 9 – p.14/37

Applicative Functors and Monads

A requirement is $\text{return} = \text{pure}$.

In fact, the *Monad* class provides a default definition of return defined that way:

class Applicative m => Monad m where

$$\begin{aligned} \text{return} &:: a \rightarrow m a \\ \text{return} &= \text{pure} \\ (\gg) &:: m a \rightarrow (a \rightarrow m b) \rightarrow m b \end{aligned}$$

COMP4075: Lecture 9 – p.17/37

Class Alternative

The class *Alternative* is a monoid on applicative functors:

class Applicative f => Alternative f where

$$\begin{aligned} \text{empty} &:: f a \\ \langle\langle| \rangle\rangle &:: f a \rightarrow f a \rightarrow f a \\ \text{some} &:: f a \rightarrow f [a] \\ \text{many} &:: f a \rightarrow f [a] \\ \text{some } v &= \text{pure } (\cdot) \langle\&*\rangle v \langle\&*\rangle \text{many } v \\ \text{many } v &= \text{some } v \langle\langle| \rangle\rangle \text{pure } [] \end{aligned}$$

$\langle\langle| \rangle\rangle$ can be understood as “one or the other”, *some* as “at least one”, and *many* as “zero or more”.

COMP4075: Lecture 9 – p.15/37

Example: Applicative Parser (2)

Syntax for a language fragment:

```
command → if expr then command else command
         | begin { command ; } end
```

Abstract syntax:

```
data Command = If Expr Command Command
              | Block [Command]
```

Recognising terminals:

```
kwd, symb :: String → Parser ()
```

COMP4075: Lecture 9 – p.15/37

Exercise: A State Monad in Haskell

Recall that a type $\text{Int} \rightarrow (a, \text{Int})$ can be viewed as a state monad.

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

```
newtype S a = S { unS :: (Int → (a, Int)) }
```

Thus: $\text{unS} :: S a \rightarrow (\text{Int} \rightarrow (a, \text{Int}))$

Provide a *Functor*, *Applicative*, and *Monad* instance for *S*.

COMP4075: Lecture 9 – p.18/37

Solution: Functor Instance

```
instance Functor S where
  fmap f sa = S $ \s →
  let
    (a, s') = unS sa s
  in
    (f a, s')
```

COMP4075: Lecture 9 – p.19/37

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)
```

COMP4075: Lecture 9 – p.20/37

Monad-specific Operations (2)

Typical operations on a state monad:

```
set :: Int → S ()
set a = S (\_ → (), a)
get :: S Int
get = S (\s → (s, s))
```

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

```
runS :: S a → a
runS m = fst (unS m 0)
```

COMP4075: Lecture 9 – p.25/37

Solution: Applicative Instance

```
instance Applicative S where
  pure a = S $ \s → (a, s)
  sf <*> sa = S $ \s →
  let
    (f, s') = unS sf s
  in
    unS (fmap f sa) s'
```

COMP4075: Lecture 9 – p.20/37

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
```

COMP4075: Lecture 9 – p.25/37

The do-notation (1)

Haskell provides convenient syntax for
programming with monads:

```
do
  a ← exp1
  b ← exp2
  return exp3
```

is syntactic sugar for

```
exp1 >>= \a →
exp2 >>= \b →
return exp3
```

Note: *a* in scope in *exp₂*, *a* and *b* in *exp₃*.

COMP4075: Lecture 9 – p.26/37

Solution: Monad Instance

```
instance Monad S where
  m >>= f = S $ \s →
  let (a, s') = unS m s
  in unS (f a) s'
```

(Using the default definition `return = pure`.)

COMP4075: Lecture 9 – p.21/37

Monad-specific Operations (1)

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

```
fail :: String → Maybe a
fail s = Nothing
catch :: Maybe a → Maybe a → Maybe a
m1 'catch' m2 =
  case m1 of
    Just _ → m1
    Nothing → m2
```

COMP4075: Lecture 9 – p.24/37

The do-notation (2)

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

```
do
  exp1
  exp2
  return exp3
```

is syntactic sugar for

```
exp1 >>= \_ →
exp2 >>= \_ →
return exp3
```

COMP4075: Lecture 9 – p.27/37

The do-notation (3)

A let-construct is also provided:

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

is equivalent to

```
do
  a ← return exp1
  b ← return exp2
  return exp3
```

COMP4075: Lecture 9 – p.28/37

Applicative do-notation (2)

For example, an applicative parser:

```
commandIf :: Parser Command
commandIf =
  kwd "if"
  c ← expr
  kwd "then"
  t ← command
  kwd "else"
  e ← command
  return (If c t e)
```

COMP4075: Lecture 9 – p.31/37

The Haskell IO Monad (2)

IO essentially provides all effects of typical imperative languages. Besides input/output:

- Pointers and imperative state (through *IORef*)
- Raising and handling exceptions
- Concurrency
- Foreign function interface

IO is sometimes referred to as the “sin bin”!

COMP4075: Lecture 9 – p.34/37

Numbering Trees in do-notation

```
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')
```

COMP4075: Lecture 9 – p.29/37

Monadic Utility Functions

Some monad utilities:

```
sequence :: Monad m ⇒ [m a] → m [a]
sequence_ :: Monad m ⇒ [m a] → m ()
mapM :: Monad m ⇒ (a → m b) → [a] → m [b]
mapM_ :: Monad m ⇒ (a → m b) → [a] → m ()
when :: Monad m ⇒ Bool → m () → m ()
foldM :: Monad m ⇒
  (a → b → m a) → a → [b] → m a
liftM :: Monad m ⇒ (a → b) → m a → m b
liftM2 :: Monad m ⇒
  (a → b → c) → m a → m b → m c
```

COMP4075: Lecture 9 – p.32/37

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:

```
data ST s a -- abstract
instance Monad (ST s)
newSTRef :: s ST a (STRef s a)
readSTRef :: STRef s a → ST s a
writeSTRef :: STRef s a → a → ST s ()
runST :: (forall s . st s a) → a
```

COMP4075: Lecture 9 – p.35/37

Applicative do-notation (1)

A variation of the do-notation is also available for applicatives:

```
do
  a ← exp1
  b ← exp2
  return (... a ... b ...)
```

Note that the bound variables may only be used in the *return*-expression, or the code becomes monadic.

In this case, *a* must not occur in *exp₂*.

COMP4075: Lecture 9 – p.30/37

The Haskell IO Monad (1)

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
getContent :: IO String
```

COMP4075: Lecture 9 – p.33/37

ST vs IO

Why use *ST* if *IO* also gives access to imperative state?

- *ST* much more focused: provides only state, not a lot more besides.
- *ST* computations can be run safely inside pure code.

It **is** possible to run *IO* comp. inside pure code:

```
unsafePerformIO :: IO a → a
```

But make sure you know what you are doing!

COMP4075: Lecture 9 – p.36/37

Reading

- Philip Wadler. The Essence of Functional Programming. *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)*, 1992.
- Nick Benton, John Hughes, Eugenio Moggi. Monads and Effects. In *International Summer School on Applied Semantics 2000*, Caminha, Portugal, 2000.