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):

$$\textbf{class } \mathit{Monad}\ m\ \textbf{where}$$
$$\quad \mathit{return} :: a \rightarrow m\ a$$
$$\quad (\ggg\!=) \ :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$$

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

Monads in Haskell (2)

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

$$(\gg) :: m\ a \rightarrow m\ b \rightarrow m\ b$$
$$m \gg k = m \ggg\!= \lambda\_ \rightarrow k$$
$$\mathit{fail} :: \mathit{String} \rightarrow m\ a$$
$$\mathit{fail}\ s = \mathit{error}\ s$$

(However, fail will likely be moved into a separate class $\mathit{MonadFail}$ in the future.)

The *Maybe* Monad in Haskell

$$
\begin{aligned}
&\mathbf{instance}\ \mathit{Monad}\ \mathit{Maybe}\ \mathbf{where} \\
&\quad \mathit{return} = \mathit{Just} \\
&\quad \mathit{Nothing} \gg\!= \_ = \mathit{Nothing} \\
&\quad (\mathit{Just}\ x) \gg\!= f = f\ x
\end{aligned}
$$

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:

$$
\begin{aligned}
&\mathbf{class}\ \mathit{Functor}\ f\ \mathbf{where} \ldots \\
&\mathbf{class}\ \mathit{Functor}\ f \Rightarrow \mathit{Applicative}\ f\ \mathbf{where} \ldots \\
&\mathbf{class}\ \mathit{Applicative}\ m \Rightarrow \mathit{Monad}\ m\ \mathbf{where} \ldots
\end{aligned}
$$

The Monad Type Class Hierachy (2)

For example, fmap can be defined in terms of $\gg\!=$ and return, demonstrating that a monad is a functor:

$$\mathit{fmap}\ f\ m = m \gg\!= \lambda x \rightarrow \mathit{return}\ (f\ x)$$

A consequence of this class hierarchy is that to make some T an instance of Monad, an instance of T for both $\mathit{Functor}$ and $\mathit{Applicative}$ 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.

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 ($\langle * \rangle$)

$$
\begin{aligned}
&\mathbf{class}\ \mathit{Functor}\ f \Rightarrow \mathit{Applicative}\ f\ \mathbf{where} \\
&\quad \mathit{pure} :: a \rightarrow f\ a \\
&\quad (\langle * \rangle) :: f\ (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b \\
&\quad (\ * \rangle) :: f\ a \rightarrow f\ b \rightarrow f\ b \\
&\quad (\langle *\ ) :: f\ a \rightarrow f\ b \rightarrow f\ a
\end{aligned}
$$

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.

Applicative Functors (3)

Laws:

$$\begin{aligned} pure\ id \ \mathbin{<\!\!*\!\!>}\ v &= v \\ pure\ (\circ) \mathbin{<\!\!*\!\!>} u \mathbin{<\!\!*\!\!>} v \mathbin{<\!\!*\!\!>} w &= u \mathbin{<\!\!*\!\!>} (v \mathbin{<\!\!*\!\!>} w) \\ pure\ f \mathbin{<\!\!*\!\!>} pure\ x &= pure\ (f\ x) \\ u \mathbin{<\!\!*\!\!>} pure\ y &= pure\ (\$y) \mathbin{<\!\!*\!\!>} u \end{aligned}$$

Default definitions:

$$\begin{aligned} u \mathbin{*\!\!>} v &= pure\ (const\ id) \mathbin{<\!\!*\!\!>} u \mathbin{<\!\!*\!\!>} v \\ u \mathbin{<\!\!*} v &= pure\ const \mathbin{<\!\!*\!\!>} u \mathbin{<\!\!*\!\!>} v \end{aligned}$$

Instances of *Applicative*

$$\begin{aligned} &\textbf{instance}\ Applicative\ [\,]\ \textbf{where} \\ &\quad pure\ x = [x] \\ &\quad fs \mathbin{<\!\!*\!\!>} xs = [f\ x \mid f \leftarrow fs, x \leftarrow xs] \end{aligned}$$

$$\begin{aligned} &\textbf{instance}\ Applicative\ Maybe\ \textbf{where} \\ &\quad pure = Just \\ &\quad Just\ f \quad \mathbin{<\!\!*\!\!>} m = fmap\ f\ m \\ &\quad Nothing \mathbin{<\!\!*\!\!>} \_ \ = Nothing \end{aligned}$$

Class *Alternative*

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

$$\begin{aligned} &\textbf{class}\ Applicative\ f \Rightarrow Alternative\ f\ \textbf{where} \\ &\quad empty :: f\ a \\ &\quad (<\mid>) :: f\ a \rightarrow f\ a \rightarrow f\ a \\ &\quad some\ \ :: f\ a \rightarrow f\ [a] \\ &\quad many :: f\ a \rightarrow f\ [a] \\ &\quad some\ v\ = pure\ (:) \mathbin{<\!\!*\!\!>} v \mathbin{<\!\!*\!\!>} many\ v \\ &\quad many\ v = some\ v <\mid> pure\ [\,] \end{aligned}$$

$<\mid>$ can be understood as “one or the other”, *some* as “at least one”, and *many* as “zero or more”.

Instances of *Alternative*

```
instance Alternative [] where
  empty = []
  (<|>) = (++)

instance Alternative Maybe where
  empty = Nothing
  Nothing <|> r = r
  l       <|> _ = l
```


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 ...
```


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 ()
```


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"
```

Applicative Functors and Monads

A requirement is $return = pure$.

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

$$\textbf{class}\ Applicative\ m \Rightarrow Monad\ m\ \textbf{where}$$
$$return :: a \to m\ a$$
$$return = pure$$
$$(\gg=) :: m\ a \to (a \to m\ b) \to m\ b$$

Exercise: A State Monad in Haskell

Recall that a type $Int \to (a, Int)$ 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:

$$\textbf{newtype}\ S\ a = S\ \{unS :: (Int \to (a, Int))\}$$

Thus: $unS :: S\ a \to (Int \to (a, Int))$

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

Solution: *Functor* Instance

$$\textbf{instance}\ Functor\ S\ \textbf{where}$$
$$fmap\ f\ sa = S\ \$\ \lambda s \to$$
$$\textbf{let}$$
$$(a, s') = unS\ sa\ s$$
$$\textbf{in}$$
$$(f\ a, s')$$

Solution: *Applicative* Instance

$$\textbf{instance}\ Applicative\ S\ \textbf{where}$$
$$pure\ a = S\ \$\ \lambda s \to (a, s)$$
$$sf \circledast sa = S\ \$\ \lambda s \to$$
$$\textbf{let}$$
$$(f, s') = unS\ sf\ s$$
$$\textbf{in}$$
$$unS\ (fmap\ f\ sa)\ s'$$

Solution: *Monad* Instance

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ S\ \textbf{where} \\
\quad m \gg\!= f = S\ \$\ \lambda s \rightarrow \\
\quad\quad \textbf{let}\ (a, s') = \mathit{unS}\ m\ s \\
\quad\quad \textbf{in}\ \mathit{unS}\ (f\ a)\ s'
\end{array}
$$

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

The List Monad

Computation with many possible results, “nondeterminism”:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ [\,]\ \textbf{where} \\
\quad \mathit{return}\ a = [a] \\
\quad m \gg\!= f = \mathit{concat}\ (\mathit{map}\ f\ m) \\
\quad \mathit{fail}\ s \quad = [\,]
\end{array}
$$

Example:

$$
\begin{array}{l}
x \leftarrow [1, 2] \\
y \leftarrow [\texttt{'a'}, \texttt{'b'}] \\
\mathit{return}\ (x, y)
\end{array}
$$

Result:

$$
\begin{array}{l}
[(1, \texttt{'a'}), (1, \texttt{'b'}), \\
(2, \texttt{'a'}), (2, \texttt{'b'})]
\end{array}
$$

The Reader Monad

Computation in an environment:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ ((\rightarrow)\ e)\ \textbf{where} \\
\quad \mathit{return}\ a = \mathit{const}\ a \\
\quad m \gg\!= f = \lambda e \rightarrow f\ (m\ e)\ e \\
\mathit{getEnv} :: ((\rightarrow)\ e)\ e \\
\mathit{getEnv} = \mathit{id}
\end{array}
$$

Monad-specific Operations (1)

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

$$
\begin{array}{l}
\mathit{fail} :: \mathit{String} \rightarrow \mathit{Maybe}\ a \\
\mathit{fail}\ s = \mathit{Nothing} \\
\mathit{catch} :: \mathit{Maybe}\ a \rightarrow \mathit{Maybe}\ a \rightarrow \mathit{Maybe}\ a \\
\mathit{m1}\ \text{`}\mathit{catch}\text{`}\ \mathit{m2} = \\
\quad \textbf{case}\ \mathit{m1}\ \textbf{of} \\
\quad\quad \mathit{Just}\ \_ \quad \rightarrow \mathit{m1} \\
\quad\quad \mathit{Nothing} \rightarrow \mathit{m2}
\end{array}
$$

Monad-specific Operations (2)

Typical operations on a state monad:

$$set :: Int \to S\ ()$$
$$set\ a = S\ (\lambda\_ \to ((), a))$$
$$get :: S\ Int$$
$$get = S\ (\lambda s \to (s, s))$$

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

$$runS :: S\ a \to a$$
$$runS\ m = fst\ (unS\ m\ 0)$$

The do-notation (1)

Haskell provides convenient syntax for programming with monads:

do
$$a \leftarrow exp_1$$
$$b \leftarrow exp_2$$
$$return\ exp_3$$

is syntactic sugar for

$$exp_1 \gg\!= \lambda a \to$$
$$exp_2 \gg\!= \lambda b \to$$
$$return\ exp_3$$

Note: a in scope in exp_2, a and b in 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

$$exp_1 \gg\!= \lambda\_ \to$$
$$exp_2 \gg\!= \lambda\_ \to$$
$$return\ exp_3$$

The do-notation (3)

A **let**-construct is also provided:

do
$$\textbf{let}\ a = exp_1$$
$$b = exp_2$$
$$return\ exp_3$$

is equivalent to

do
$$a \leftarrow return\ exp_1$$
$$b \leftarrow return\ exp_2$$
$$return\ exp_3$$

Numbering Trees in do-notation

$$
\begin{array}{l}
\mathit{numberTree}\ t = \mathit{runS}\ (\mathit{ntAux}\ t)\\
\quad \textbf{where}\\
\qquad \mathit{ntAux} :: \mathit{Tree}\ a \to S\ (\mathit{Tree}\ \mathit{Int})\\
\qquad \mathit{ntAux}\ (\mathit{Leaf}\ \_) = \textbf{do}\\
\qquad\quad n \leftarrow \mathit{get}\\
\qquad\quad \mathit{set}\ (n + 1)\\
\qquad\quad \mathit{return}\ (\mathit{Leaf}\ n)\\
\qquad \mathit{ntAux}\ (\mathit{Node}\ t1\ t2) = \textbf{do}\\
\qquad\quad t1' \leftarrow \mathit{ntAux}\ t1\\
\qquad\quad t2' \leftarrow \mathit{ntAux}\ t2\\
\qquad\quad \mathit{return}\ (\mathit{Node}\ t1'\ t2')
\end{array}
$$

Applicative do-notation (1)

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

$$
\begin{array}{l}
\textbf{do}\\
\quad a \leftarrow \mathit{exp}_1\\
\quad b \leftarrow \mathit{exp}_2\\
\quad \mathit{return}\ (\ldots a \ldots b \ldots)
\end{array}
$$

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 exp_2.

Applicative do-notation (2)

For example, an applicative parser:

$$
\begin{array}{l}
\mathit{commandIf} :: \mathit{Parser}\ \mathit{Command}\\
\mathit{commandIf} =\\
\quad \mathit{kwd}\ \texttt{"if"}\\
\quad c \leftarrow \mathit{expr}\\
\quad \mathit{kwd}\ \texttt{"then"}\\
\quad t \leftarrow \mathit{command}\\
\quad \mathit{kwd}\ \texttt{"else"}\\
\quad e \leftarrow \mathit{command}\\
\quad \mathit{return}\ (\mathit{If}\ c\ t\ e)
\end{array}
$$

Monadic Utility Functions

Some monad utilities:

$$
\begin{array}{ll}
\mathit{sequence} & :: \mathit{Monad}\ m \Rightarrow [m\ a] \to m\ [a]\\
\mathit{sequence\_} & :: \mathit{Monad}\ m \Rightarrow [m\ a] \to m\ ()\\
\mathit{mapM} & :: \mathit{Monad}\ m \Rightarrow (a \to m\ b) \to [a] \to m\ [b]\\
\mathit{mapM\_} & :: \mathit{Monad}\ m \Rightarrow (a \to m\ b) \to [a] \to m\ ()\\
\mathit{when} & :: \mathit{Monad}\ m \Rightarrow \mathit{Bool} \to m\ () \to m\ ()\\
\mathit{foldM} & :: \mathit{Monad}\ m \Rightarrow\\
 & \quad (a \to b \to m\ a) \to a \to [b] \to m\ a\\
\mathit{liftM} & :: \mathit{Monad}\ m \Rightarrow (a \to b) \to m\ a \to m\ b\\
\mathit{liftM2} & :: \mathit{Monad}\ m \Rightarrow\\
 & \quad (a \to b \to c) \to m\ a \to m\ b \to m\ c
\end{array}
$$

The Haskell IO Monad (1)

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

$$\mathbf{newtype}\ IO\ a = IO\ (World \to (a, World))$$

Some operations:

$$\begin{aligned}
&putChar && :: Char \to IO\ ()\\
&putStr && :: String \to IO\ ()\\
&putStrLn && :: String \to IO\ ()\\
&getChar && :: IO\ Char\\
&getLine && :: IO\ String\\
&getContents && :: IO\ String
\end{aligned}$$

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"!

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:

$\mathbf{data}\ ST\ s\ a$ -- abstract

$\mathbf{instance}\ Monad\ (ST\ s)$

$$\begin{aligned}
&newSTRef && :: s\ ST\ a\ (STRef\ s\ a)\\
&readSTRef && :: STRef\ s\ a \to ST\ s\ a\\
&writeSTRef && :: STRef\ s\ a \to a \to ST\ s\ ()
\end{aligned}$$

$$runST :: (forall\ s\ .\ st\ s\ a) \to a$$

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:

$$unsafePerformIO :: IO\ a \to a$$

But make sure you know what you are doing!

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.