COMP4075/G54RFP: Lecture 8

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):

$$
\begin{array}{l}
\textbf{class}\ Monad\ m\ \textbf{where} \\
\quad return :: a \to m\ a \\
\quad (\gg\!=) \ \ :: m\ a \to (a \to m\ b) \to m\ b
\end{array}
$$

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:

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

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

The *Maybe* Monad in Haskell

$$
\begin{array}{l}
\textbf{instance } \mathit{Monad}\ \mathit{Maybe}\ \textbf{where} \\
\quad \mathit{return} = \mathit{Just} \\
\quad \mathit{Nothing} \gg\!= \_ = \mathit{Nothing} \\
\quad (\mathit{Just}\ x) \gg\!= f = f\ x
\end{array}
$$

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:

$$\textbf{class}\ \mathit{Functor}\ f\ \textbf{where} \ldots$$
$$\textbf{class}\ \mathit{Functor}\ f \Rightarrow \mathit{Applicative}\ f\ \textbf{where} \ldots$$
$$\textbf{class}\ \mathit{Applicative}\ m \Rightarrow \mathit{Monad}\ m\ \textbf{where} \ldots$$

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:

$$fmap\ f\ \ m = m \gg= \lambda x \rightarrow return\ (f\ \ x)$$

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:

$$fmap\ f\ m = m \gg\!= \lambda x \to return\ (f\ x)$$

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.

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:

$$fmap\ f\ m = m \gg= \lambda x \rightarrow return\ (f\ x)$$

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.

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 ($\circledast$)

$$
\begin{array}{l}
\mathbf{class}\ Functor\ f \Rightarrow Applicative\ f\ \mathbf{where} \\
\quad \begin{array}{ll}
pure & :: a \to f\ a \\
(\circledast) & :: f\ (a \to b) \to f\ a \to f\ b \\
(\ \ast\!\!>) & :: f\ a \to f\ b \to f\ b \\
(<\!\!\ast\ ) & :: f\ a \to f\ b \to f\ a
\end{array}
\end{array}
$$

Applicative Functors (2)

- Like monads, applicative functors is a notion of computation.

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***.

Applicative Functors (3)

Laws:

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

Applicative Functors (3)

Laws:

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

Default definitions:

$$u \mathrel{*\!\!\gt} v = pure\ (const\ id) \lt\!\!*\!\!\gt u \lt\!\!*\!\!\gt v$$

$$u \mathrel{\lt\!\!*} v = pure\ const \lt\!\!*\!\!\gt u \lt\!\!*\!\!\gt v$$

Instances of *Applicative*

$$\textbf{instance}\ \mathit{Applicative}\ [\,]\ \textbf{where}$$
$$\quad \mathit{pure}\ x = [x]$$
$$\quad \mathit{fs} \circledast \mathit{xs} = [f\ x \mid f \leftarrow \mathit{fs}, x \leftarrow \mathit{xs}]$$

Instances of $Applicative$

$$
\begin{array}{l}
\textbf{instance}\ Applicative\ [\,]\ \textbf{where} \\
\quad pure\ x = [x] \\
\quad fs \circledast xs = [f\ x \mid f \leftarrow fs, x \leftarrow xs]
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ Applicative\ Maybe\ \textbf{where} \\
\quad pure = Just \\
\quad Just\ f \quad \circledast m = fmap\ f\ m \\
\quad Nothing \circledast \_ \ = Nothing
\end{array}
$$

Class *Alternative*

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

$$
\begin{array}{l}
\textbf{class } \mathit{Applicative}\ f \Rightarrow \mathit{Alternative}\ f\ \textbf{where} \\
\quad \mathit{empty} :: f\ a \\
\quad (<\!|\!>) :: f\ a \to f\ a \to f\ a \\
\quad \mathit{some}\ \ :: f\ a \to f\ [a] \\
\quad \mathit{many}\ :: f\ a \to f\ [a] \\
\quad \mathit{some}\ v\ = \mathit{pure}\ (:) <\!\!*\!\!> v <\!\!*\!\!> \mathit{many}\ v \\
\quad \mathit{many}\ v = \mathit{some}\ v <\!|\!> \mathit{pure}\ [\,]
\end{array}
$$

Class *Alternative*

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

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

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

Instances of *Alternative*

$$
\begin{array}{l}
\textbf{instance } \mathit{Alternative}\ [\,]\ \textbf{where} \\
\quad \mathit{empty} = [\,] \\
\quad (<\!|\!>) = (+\!\!+)
\end{array}
$$

Instances of *Alternative*

$$
\begin{array}{l}
\textbf{instance } \mathit{Alternative}\ [\,]\ \textbf{where} \\
\quad \mathit{empty} = [\,] \\
\quad (\mathopen{<}|\mathclose{>}) = (+\!\!+)
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance } \mathit{Alternative}\ \mathit{Maybe}\ \textbf{where} \\
\quad \mathit{empty} = \mathit{Nothing} \\
\quad \mathit{Nothing} \mathbin{<|>} r = r \\
\quad l \qquad\qquad\ \mathbin{<|>} \_ = l
\end{array}
$$

Example: Applicative Parser (1)

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

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 and trying alternatives:

$$\textbf{instance}\ \mathit{Applicative}\ \mathit{Parser}\ \textbf{where} \ldots$$
$$\textbf{instance}\ \mathit{Alternative}\ \mathit{Parser}\ \textbf{where} \ldots$$

Example: Applicative Parser (2)

$$
\begin{array}{l}
\mathit{command} :: \mathit{Parser}\ \mathit{Command} \\
\mathit{command} = \\
\qquad\qquad \mathit{pure}\ \mathit{If} \\
\qquad\qquad\ {<\!\!*}\quad \mathit{kwd}\ \texttt{"if"} \qquad {<\!\!*\!\!>}\ \mathit{expr} \\
\qquad\qquad\ {<\!\!*}\quad \mathit{kwd}\ \texttt{"then"}\ {<\!\!*\!\!>}\ \mathit{command} \\
\qquad\qquad\ {<\!\!*}\quad \mathit{kwd}\ \texttt{"else"}\ {<\!\!*\!\!>}\ \mathit{command} \\
\qquad <\!|\!>\ \mathit{pure}\ \mathit{Block} \\
\qquad\qquad\ {<\!\!*}\quad \mathit{kwd}\ \texttt{"begin"} \\
\qquad\qquad\ {<\!\!*\!\!>}\ \mathit{some}\ (\mathit{command}\ {*\!\!>}\ \mathit{symb}\ \texttt{";"}) \\
\qquad\qquad\ {<\!\!*}\quad \mathit{kwd}\ \texttt{"end"}
\end{array}
$$

$\ldots$

Applicative Functors and Monads

A requirement is $return = pure$.

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

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

Exercise: A State Monad in Haskell

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

$$\mathbf{newtype}\ S\ a = S\ \{ unS :: (Int \rightarrow (a, Int)) \}$$

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

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

Solution: *Functor* Instance

$$
\begin{array}{l}
\textbf{instance } \mathit{Functor}\ S \textbf{ where} \\
\quad \mathit{fmap}\ f\ \ sa = S\ \$\ \lambda s \rightarrow \\
\qquad \textbf{let} \\
\qquad\quad (a, s') = \mathit{unS}\ \ sa\ \ s \\
\qquad \textbf{in} \\
\qquad\quad (f\ \ a, s')
\end{array}
$$

Solution: *Applicative* Instance

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

Solution: *Monad* Instance

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

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

The List Monad

Computation with many possible results, "nondeterminism":

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

Example:

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

Result:

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

The Reader Monad

Computation in an environment:

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

$$
\begin{array}{l}
\mathit{getEnv} :: ((\rightarrow)\ e)\ e \\
\mathit{getEnv} = \mathit{id}
\end{array}
$$

Monad-specific Operations (1)

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

$$fail :: String \to Maybe\ a$$
$$fail\ s = Nothing$$

$$catch :: Maybe\ a \to Maybe\ a \to Maybe\ a$$
$$\begin{aligned} & m1\ \text{`}catch\text{`}\ m2 = \\ & \quad \textbf{case}\ m1\ \textbf{of} \\ & \qquad Just\ \_ \quad \to m1 \\ & \qquad Nothing \to m2 \end{aligned}$$

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:

$$
\begin{array}{l}
\mathbf{do} \\
\quad a \leftarrow exp_1 \\
\quad b \leftarrow exp_2 \\
\quad return\ exp_3
\end{array}
$$

is syntactic sugar for

$$
\begin{array}{l}
exp_1 \gg\!= \lambda a \rightarrow \\
exp_2 \gg\!= \lambda b \rightarrow \\
return\ exp_3
\end{array}
$$

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:

$$
\begin{array}{l}
\textbf{do} \\
\quad exp_1 \\
\quad exp_2 \\
\quad return \;\; exp_3
\end{array}
$$

is syntactic sugar for

$$
\begin{array}{l}
exp_1 \gg\!= \lambda\_ \rightarrow \\
exp_2 \gg\!= \lambda\_ \rightarrow \\
return \;\; exp_3
\end{array}
$$

The do-notation (3)

A $\mathbf{let}$-construct is also provided:

$$
\begin{array}{l}
\mathbf{do} \\
\quad \mathbf{let}\ a = exp_1 \\
\quad \phantom{\mathbf{let}\ } b = exp_2 \\
\quad return\ exp_3
\end{array}
$$

is equivalent to

$$
\begin{array}{l}
\mathbf{do} \\
\quad a \leftarrow return\ exp_1 \\
\quad b \leftarrow return\ exp_2 \\
\quad return\ exp_3
\end{array}
$$

Numbering Trees in do-notation

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

Applicative do-notation (1)

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

$$
\begin{array}{l}
\mathbf{do} \\
\quad a \leftarrow exp_1 \\
\quad b \leftarrow exp_2 \\
\quad return\ (\ldots a \ldots b \ldots)
\end{array}
$$

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:

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

The Haskell IO Monad (1)

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

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

Some operations:

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

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

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:

$$\textbf{data}\ ST\ s\ a \quad \text{-- abstract}$$

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

$$newSTRef\ ::\ s\ ST\ a\ (STRef\ s\ a)$$

$$readSTRef\ ::\ STRef\ s\ a \to ST\ s\ a$$

$$writeSTRef\ ::\ STRef\ s\ a \to a \to ST\ s\ ()$$

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

ST vs IO

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

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 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.

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:

$$unsafePerformIO :: IO\ a \rightarrow a$$

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.