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):

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

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 $\ggg=$ and $return$, demonstrating that a monad is a functor:

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

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:

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

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.

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 ($\circledast$)

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

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:

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.

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 (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:

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

Applicative Functors (3)

Laws:

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

Default definitions:

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

Instances of $Applicative$

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

Instances of $Applicative$

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

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

Class $\mathit{Alternative}$

The class $\mathit{Alternative}$ 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 $\mathit{Alternative}$

The class $\mathit{Alternative}$ is a monoid on applicative functors:

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

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

Instances of *Alternative*

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

Instances of *Alternative*

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

$$\textbf{instance}\ \mathit{Alternative}\ \mathit{Maybe}\ \textbf{where}$$
$$\quad \mathit{empty} = \mathit{Nothing}$$
$$\quad \mathit{Nothing} <\!|\!> r = r$$
$$\quad l \quad <\!|\!> \_ = l$$

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, fails if input cannot be parsed, and supports trying alternatives:

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

Example: Applicative Parser (2)

Syntax for a language fragment:

$$\begin{array}{lcl} \mathit{command} & \rightarrow & \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{command}\ \texttt{else}\ \mathit{command} \\ & \mid & \texttt{begin}\ \{\ \mathit{command}\ \texttt{;}\ \}\ \texttt{end} \end{array}$$

Abstract syntax:

$$\begin{array}{l} \textbf{data}\ \mathit{Command} = \mathit{If}\ \mathit{Expr}\ \mathit{Command}\ \mathit{Command} \\ \qquad\qquad\qquad\quad \mid\ \mathit{Block}\ [\mathit{Command}] \end{array}$$

Recognising terminals:

$$\mathit{kwd}, \mathit{symb} :: \mathit{String} \rightarrow \mathit{Parser}\ ()$$

Example: Applicative Parser (3)

$$
\begin{array}{l}
command :: Parser\ Command \\
command = \\
\qquad\qquad pure\ If \\
\qquad\qquad\ \ \lessdot\!*\ \ kwd\ \texttt{"if"}\qquad \lessdot\!*\!\gtrdot\ expr \\
\qquad\qquad\ \ \lessdot\!*\ \ kwd\ \texttt{"then"}\ \lessdot\!*\!\gtrdot\ command \\
\qquad\qquad\ \ \lessdot\!*\ \ kwd\ \texttt{"else"}\ \lessdot\!*\!\gtrdot\ command \\
\qquad \lessdot\!|\!\gtrdot\ pure\ Block \\
\qquad\qquad\ \ \lessdot\!*\ \ kwd\ \texttt{"begin"} \\
\qquad\qquad\ \ \lessdot\!*\!\gtrdot\ many\ (command \lessdot\!*\ symb\ \texttt{";"}) \\
\qquad\qquad\ \ \lessdot\!*\ \ kwd\ \texttt{"end"}
\end{array}
$$

Applicative Functors and Monads

A requirement is $return = pure$.

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

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

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:

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

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

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

Solution: *Functor* Instance

instance *Functor* S **where**

$fmap\ f\ \ sa = S\ \$\ \lambda s \rightarrow$

let

$(a, s') = unS\ \ sa\ \ s$

in

$(f\ \ a, s')$

Solution: *Applicative* Instance

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

Solution: *Monad* Instance

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

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

The List Monad

Computation with many possible results, "nondeterminism":

$$\textbf{instance}\ Monad\ [\,]\ \textbf{where}$$

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

Example:

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

Result:

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

The Reader Monad

Computation in an environment:

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

Monad-specific Operations (1)

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

$$
\begin{aligned}
&\mathit{fail} :: \mathit{String} \to \mathit{Maybe}\ a\\
&\mathit{fail}\ s = \mathit{Nothing}
\end{aligned}
$$

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

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:

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

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 let-construct is also provided:

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

is equivalent to

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

Numbering Trees in do-notation

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

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:

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

Monadic Utility Functions

Some monad utilities:

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

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

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:

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

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

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

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

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

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

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.