Programming with Monads

Roshan Shariff

January 8, 2013

Introduction

What are Monads?

A monad is ...

- ... like a spacesuit
- ... like a burrito
- ... a monster that devours values
- ... a monoid in the category of endofunctors. What's the problem?[1]

[1] *A Brief, Incomplete, and Mostly Wrong History of Programming Languages* by James Iry.

Really, What are Monads?

Monads are

- a pattern for designing software libraries having
  - a family of types
  - functions that operate on those types
- a way to define the semantics of programs ...
  ... by defining them using primitive *computations* combined together

Monads are *not*

- a built-in language feature of Haskell
- a way to sneak side effects into a pure language
- *just* a way to perform input/output in a pure functional language
- a one-size-fits-all solution to designing combinator libraries; comonads, arrows, (applicative) functors, etc. might be better

Monadic Values

Suppose M is a monad (a software library with a monadic interface). The monadic types are:

- M `Integer`
- M `String`
- M $()$
- $\ldots$
- $M\ t$ for any type t

M is a *type constructor*

Any $\mathtt{x} :: M\ t$ is called a *monadic value* of type t
(think of it as a *computation* that produces a t value)

To construct monadic values, there must be a function

$$\mathtt{unit} :: a \to M\ a \qquad \text{(aka } \mathtt{return}, \mathtt{pure}\text{)}$$

Monadic Functions

Any $\mathtt{f} :: a \to M\ b$ is called a *monadic function* from a to b

$\mathtt{unit} :: a \to M\ a$ is a monadic function from a to a.

Any function $\mathtt{f} :: a \to b$ can be turned into a monadic function by composing it with $\mathtt{unit}$.

$$\mathtt{fM} :: a \to M\ b$$

$$\mathtt{fM} = \mathtt{unit} \circ \mathtt{f} \qquad \text{equivalently} \quad \mathtt{fM}\ \mathtt{x} = \mathtt{unit}\ (\mathtt{f}\ \mathtt{x})$$

because $\mathtt{unit}\ \mathtt{x}$ is a ‘trivial’ computation: it does nothing but produce $\mathtt{x}$ (we will see what this means later)

The Bind Combinator

How do we ‘apply’ a monadic function to a monadic value?

$$\texttt{x} :: M\ a \qquad \text{(a monadic value)}$$

$$\texttt{f} :: a \to M\ b \qquad \text{(a monadic function)}$$

Since `f x` does not work, the monadic library must provide another operation,

$$\texttt{bind} :: M\ a \to (a \to M\ b) \to M\ b$$

Often `bind x f` is written infix-style as `x` \``bind`\` `f` or symbolically as $\texttt{x} \gg\!= \texttt{f}$ (that’s >>= in ASCII)

Note that this definition of `bind` does not allow pure values to ‘escape’ from monadic values (you can’t get an a from an $M\ a$)

So What are Monads?

A monad is a type constructor M along with at least two functions

$$\texttt{unit} :: a \to M\ a$$
$$\texttt{bind} :: M\ a \to (a \to M\ b) \to M\ b$$

whose implementations define the computational 'meaning' of the monad.

- `unit` creates 'trivial' primitive computations that just return a value
- Any non-trivial monad has other primitive computations that do something meaningful
- `bind` combines computations together, using the value of the first to influence what the second does

The Identity Monad

Motivation

The Identity Monad is the ‘trivial’ monad

Monadic values are just normal values. `unit` is essentially the identity function.

Monadic functions are just normal functions. `bind` is essentially function application.

There are no other primitive computations. There is no computational meaning beyond pure functions being applied to values.

A simple example to start understanding monads

Definition

$$\mathbf{data}\ \mathtt{Identity}\ t = \mathtt{Just}\ t$$

$$\mathtt{unit} :: a \to \mathtt{Identity}\ a$$
$$\mathtt{bind} :: \mathtt{Identity}\ a \to (a \to \mathtt{Identity}\ b) \to \mathtt{Identity}\ b$$

$$\mathtt{unit}\ \mathtt{x} = \mathtt{Just}\ \mathtt{x}$$
$$\mathtt{bind}\ (\mathtt{Just}\ \mathtt{x})\ \mathtt{f} = \mathtt{f}\ \mathtt{x}$$

An Example

Suppose we want to implement $f(x, y) = \sqrt{x} + \sqrt{y}$. The pure version would be

$$\mathtt{f} :: \mathtt{Double} \to \mathtt{Double} \to \mathtt{Double}$$
$$\mathtt{f}\ \mathtt{x}\ \mathtt{y} = (\mathtt{sqrt}\ \mathtt{x}) + (\mathtt{sqrt}\ \mathtt{y})$$

If we want to use the monadic square root function instead

$$\mathtt{sqrtM} :: \mathtt{Double} \to M\ \mathtt{Double}$$
$$\mathtt{sqrtM} = \mathtt{unit} \circ \mathtt{sqrt}$$

we can write a monadic version of `f` as

$$\mathtt{fM} :: \mathtt{Double} \to \mathtt{Double} \to M\ \mathtt{Double}$$
$$\begin{aligned}\mathtt{fM}\ \mathtt{x}\ \mathtt{y} = {} & \mathtt{sqrtM}\ \mathtt{x} \gg\!= \lambda\mathtt{sqrtX} \to \\ & \mathtt{sqrtM}\ \mathtt{y} \gg\!= \lambda\mathtt{sqrtY} \to \\ & \mathtt{unit}\ (\mathtt{sqrtX} + \mathtt{sqrtY})\end{aligned}$$

Some Questions

Are there any restrictions on what `unit` and `bind` are allowed to do?

Why is it okay to compose any pure function with `unit` without unexpected results?

Interlude: Monad Laws

Composing Monadic Functions

We can 'apply' a monadic function to a monadic value with

$$\texttt{bind} :: M\ a \to (a \to M\ b) \to M\ b$$

We can also use it to define the composition of two monadic functions:

$$\ggg\!\!\!\Rightarrow :: (a \to M\ b) \to (b \to M\ c) \to (a \to M\ c)$$
$$\texttt{f} \ggg\!\!\!\Rightarrow \texttt{g} = \lambda\texttt{x} \to \texttt{f x} \gg\!= \texttt{g}$$

Compare this with the pure function composition operator

$$\ggg :: (a \to b) \to (b \to c) \to (a \to c)$$
$$\texttt{f} \ggg \texttt{g} = \lambda\texttt{x} \to \texttt{g (f x)}$$

The Monad Laws

The monad laws formalize the expectation that $\ggg$ behaves like regular function composition and `unit` behaves like an identity function

Let $\mathtt{f} :: a \to M\ b$, $\mathtt{g} :: b \to M\ c$, and $\mathtt{h} :: c \to M\ d$ be monadic functions. Then

- `unit` must be an identity of $\ggg$, i.e.

$$\mathtt{unit} \ggg \mathtt{f} \equiv \mathtt{f}$$
$$\mathtt{f} \ggg \mathtt{unit} \equiv \mathtt{f}$$

- $\ggg$ must be associative, i.e.

$$(\mathtt{f} \ggg \mathtt{g}) \ggg \mathtt{h} \equiv \mathtt{f} \ggg (\mathtt{g} \ggg \mathtt{h})$$

If these laws aren't satisfied, M is not a monad.

Consequences

For any $\mathtt{x} :: a$ and monadic function $\mathtt{f} :: a \to M\ b$

$$\mathtt{unit\ x} \gg\!= \mathtt{f} \equiv \mathtt{f\ x}$$

In particular, for any $\mathtt{x} :: a$ and pure function $\mathtt{f} :: a \to b$

$$\mathtt{unit\ x} \gg\!= \mathtt{unit} \circ \mathtt{f} \equiv \mathtt{unit}\ (\mathtt{f\ x})$$

Any sequence of computations composed together

$$f \ggg g \ggg h$$

is well-defined even even without parentheses to indicate order of operations.

The Maybe Monad

Motivation

Captures the notion of computations that may fail to return a value

Monadic values are either normal values, or a special value indicating failure

A failed computation bound to any other computation causes the entire computation to fail

The interface consists of

- the `Maybe` type constructor
- the usual `unit` and `bind`
- `mzero`, a monadic value that represents a failed computation of any type[2]

[2] The name `mzero` is used to represent failure in any monad that supports it

Definition

$$\mathbf{data}\ \mathtt{Maybe}\ t = \mathtt{Just}\ t \mid \mathtt{Nothing}$$

$$\mathtt{unit} :: a \to \mathtt{Maybe}\ a$$
$$\mathtt{bind} :: \mathtt{Maybe}\ a \to (a \to \mathtt{Maybe}\ b) \to \mathtt{Maybe}\ b$$

$$\mathtt{unit}\ \mathtt{x} = \mathtt{Just}\ \mathtt{x}$$
$$\mathtt{bind}\ (\mathtt{Just}\ \mathtt{x})\ \mathtt{f} = \mathtt{f}\ \mathtt{x}$$
$$\mathtt{bind}\ \mathtt{Nothing}\ \mathtt{f} = \mathtt{Nothing}$$

$$\mathtt{mzero} :: \mathtt{Maybe}\ a$$
$$\mathtt{mzero} = \mathtt{Nothing}$$

An Example

Consider the example $f(x, y) = \sqrt{x} + \sqrt{y}$ from before. Suppose we want the `sqrtM` function to succeed only for non-negative arguments. We can define it as

$$\mathtt{sqrtM} :: \mathtt{Double} \to \mathtt{Maybe}\ \mathtt{Double}$$
$$\mathtt{sqrtM}\ \mathtt{x} = \mathbf{if}\ \mathtt{x} \geq 0\ \mathbf{then}\ \mathtt{unit}\ (\mathtt{sqrt}\ \mathtt{x})\ \mathbf{else}\ \mathtt{mzero}$$

With this change to `sqrtM`, we can use exactly the same definition of `fM` as before

$$\mathtt{fM} :: \mathtt{Double} \to \mathtt{Double} \to \mathtt{Maybe}\ \mathtt{Double}$$
$$\begin{aligned} \mathtt{fM}\ \mathtt{x}\ \mathtt{y} = {} & \mathtt{sqrtM}\ \mathtt{x} \gg\!= \lambda\mathtt{sqrtX} \to \\ & \mathtt{sqrtM}\ \mathtt{y} \gg\!= \lambda\mathtt{sqrtY} \to \\ & \mathtt{unit}\ (\mathtt{sqrtX} + \mathtt{sqrtY}) \end{aligned}$$

$\mathtt{fM}\ \mathtt{x}\ \mathtt{y}$ evaluates to $\mathtt{Just}\ (\sqrt{\mathtt{x}} + \sqrt{\mathtt{y}})$ if both $\mathtt{x}$ and $\mathtt{y}$ are non-negative, and `Nothing` otherwise.

The State Monad

Motivation

Suppose we have a function whose value depends on some state of type s (that it modifies). The signature

$$\mathtt{f} :: a \to b$$

does not fully represent the behaviour of the function, because the output of type b doesn't just depend on the input of type a.

The usual representation in a functional language is to explicitly indicate the extra s input, and return the modified state

$$\mathtt{f} :: a \to s \to (b, s)$$

Explicitly managing state is difficult and error-prone, but if we write $\mathtt{State}\ s\ t$ for $s \to (t, s)$ then `f` becomes a monadic function!

$$f :: a \to \mathtt{State}\ s\ b$$

Is State a Monad?

No.

`State` itself is not a monad, but `State` s is a monad for any fixed s.

`State` is an entire family of monads: `State Int`, `State String`, etc.

A monadic value `x` :: `State` s t is called a state transformer; it takes an initial state of type s and produces a value of type t and a new state. We can think of it as having the type

$$\mathtt{x} :: s \to (a, s)$$

Definition

$$\textbf{type}\ \mathtt{State}\ s\ t = s \to (t, s)$$
$$\mathtt{unit} :: a \to \mathtt{State}\ s\ a$$
$$\mathtt{bind} :: \mathtt{State}\ s\ a \to (a \to \mathtt{State}\ s\ b) \to \mathtt{State}\ s\ b$$

$$\mathtt{unit}\ \mathtt{x} = \lambda\mathtt{s} \to (\mathtt{x}, \mathtt{s})$$
$$\mathtt{bind}\ \mathtt{st}\ \mathtt{f} = \lambda\mathtt{s}_0 \to \textbf{let}\ (\mathtt{x}, \mathtt{s}_1) = \mathtt{st}\ \mathtt{s}_0\ \textbf{in}\ \mathtt{f}\ \mathtt{x}\ \mathtt{s}_1$$

We need a monadic value that represents the current state:

$$\mathtt{getState} :: \mathtt{State}\ s\ s$$
$$\mathtt{getState} = \lambda\mathtt{s} \to (\mathtt{s}, \mathtt{s})$$

We need a monadic function that sets a new state:

$$\mathtt{setState} :: \mathtt{s} \to \mathtt{State}\ s\ ()$$
$$\mathtt{setState}\ \mathtt{s}_1 = \lambda\mathtt{s}_0 \to ((), \mathtt{s}_1)$$

Definition (2)

Most monads don't let you extract a pure value of type t from a monadic value of type $M\ t$.

The state monad does allow this, but only if you provide an initial state

$$\mathtt{runState} :: \mathtt{State}\, s\, t \rightarrow s \rightarrow t$$
$$\mathtt{runState\ st\ s_0} = \mathbf{let}\ (\mathtt{x}, \mathtt{s_1}) = \mathtt{st\ s_0}\ \mathbf{in}\ \mathtt{x}$$

runs the provided initial state $\mathtt{s_0}$ through the monadic value (i.e. state transformer) $\mathtt{st}$ and returns the result, discarding the final state.

An Example

We have a tree data type: $\mathbf{data}\ \mathtt{Tree}\ t = \mathtt{Empty} \mid \mathtt{Node}\ (\mathtt{Tree}\ t)\ t\ (\mathtt{Tree}\ t)$

We want to traverse the tree in depth-first order and sequentially number each node.

If only we could use a single `Int` global variable as a counter...

$$
\begin{aligned}
&\mathtt{nextLabel} :: \mathtt{State}\ \mathtt{Int}\ \mathtt{Int} \\
&\mathtt{nextLabel} = \mathtt{getState} \ggg\!= \lambda\mathtt{counter} \rightarrow \\
&\qquad\qquad\quad \mathtt{setState}\ (\mathtt{counter} + 1) \ggg\!= \lambda\_ \rightarrow \\
&\qquad\qquad\quad \mathtt{unit}\ \mathtt{counter}
\end{aligned}
$$

is a monadic value that increments the counter value and returns a unique label each time it is evaluated.

An Example (Contd.)

Then the relabeling function can be written as

$$
\begin{aligned}
\mathtt{relabel}' &:: \mathtt{Tree}\ a \to \mathtt{State\ Int}\ (\mathtt{Tree\ Int}) \\
\mathtt{relabel}'\ \mathtt{Empty} &= \mathtt{unit\ Empty} \\
\mathtt{relabel}'\ (\mathtt{Node\ l\ \_\ r}) &= \mathtt{relabel}'\ \mathtt{l} \ggg\!= \lambda\mathtt{l} \to \\
&\quad\ \mathtt{nextLabel} \ggg\!= \lambda\mathtt{x} \to \\
&\quad\ \mathtt{relabel}'\ \mathtt{r} \ggg\!= \lambda\mathtt{r} \to \\
&\quad\ \mathtt{unit}\ (\mathtt{Node\ l\ x\ r})
\end{aligned}
$$

and our final function is

$$
\begin{aligned}
&\mathtt{relabel} :: \mathtt{Tree}\ a \to \mathtt{Tree\ Int} \\
&\mathtt{relabel\ tree} = \mathtt{runState}\ (\mathtt{relabel}'\ \mathtt{tree})\ 0
\end{aligned}
$$

A Special Kind of State

How do we change state that *really* matters? How do we change the state of the world?

If we had a data type `RealWorld`, then we could use the `State RealWorld` monad (aka the `IO` monad). If only...

$$\texttt{getChar} :: \texttt{IO Char}$$

$$\texttt{putChar} :: \texttt{Char} \rightarrow \texttt{IO} \; ()$$

This actually works! By hiding away the `getState` and `putState` computations, we disallow direct access to `RealWorld` (which can be just a token type).

The only actions in the `IO` monad are those that affect the outside world in some way.

We can't actually *run* `IO` monadic values, but we can construct them, and pass them to the runtime system as $\texttt{main} :: \texttt{IO} \; ()$.

Further Reading

"Monads for functional programming", *Philip Wadler* (1992)

"All About Monads" (Haskell wiki)

"Monads as containers" and "Monads as computation", *Cale Gibbard* (Haskell wiki)

"Notions of computation and monads", *Eugenio Moggi* (1991)