Monoids and Monads

George Giorgidze

Universität Tübingen

Advanced Functional Programming (Lecture 9)
(Some slides use material developed by Graham Hutton)
2011 DEC 05, Tübingen, Germany

Your feedback

- Your feedback is welcome:
  - via email
  - via EvaSys (anonymous)
  - we will respond timely and adjust the course if required
- Your feedback so far:
  - Lecture room
  - Grading scheme for Bachelor, Diploma and Masters
  - Lots of information for a single lecture
  - Course work load
  - Feedback on exercises
  - Paper/tutorial submission deadline (has been moved to 2010-Jan-15)
  - Related features in other languages
  - Monads earlier on

Plagiarism

- Benefits and pitfalls of working in teams
- Collaboration/discussion vs. copy/paste
- Coauthor responsibilities

The AFP course so far

- Pure functional programming
- Typed programs with type inference
- Polymorphism
- Overloading with type classes
- Functions as first-class values
- Higher-order combinators
- Abstractions (e.g., *Functor* and *Applicative*)
- Lazy evaluation
- Abstract data types
- Automated property-based testing

The rest of the AFP course

- Monoids and monads
- Combining monads
- Functional programming for parallelism and concurrency
- Developing purely functional data structures
- Embedded Domain-specific Languages (EDSLs)
- Generic programming
- Topics for further research/professional development (more types, type-level programming, functional programming in industry and academia, etc.)

Monoids

A *monoid* is a set, S together with a binary operation $\bullet$ that satisfies the following three axioms:

- Closure: $\forall a, b \in S : a \bullet b \in S$
- Associativity: $\forall a, b, c \in S : (a \bullet b) \bullet c = a \bullet (b \bullet c)$
- Identity element: $\exists e \in S : \forall a \in S : e \bullet a = a \bullet e = a$

Examples:

- $(\mathbb{N}, +, 0)$
- $(\mathbb{N}, *, 1)$

Monoids in Haskell

The *Monoid* type class:

```
class Monoid a where
  mempty :: a
  mappend :: a → a → a
  mconcat :: [a] → a
  mconcat = foldr mappend mempty
```

The *Monoid* laws:

$$mappend\ mempty\ x = x$$

$$mappend\ x\ mempty = x$$

$$mappend\ x\ (mappend\ y\ z) = mappend\ (mappend\ x\ y)\ z$$

Monoid Instances

$$\textbf{instance}\ \mathit{Monoid}\ [a]\ \textbf{where}$$
$$\quad \mathit{mempty} = [\,]$$
$$\quad \mathit{mappend} = (+\!\!+)$$

$$\textbf{import}\ \mathit{Data.Set}$$
$$\textbf{instance}\ \mathit{Ord}\ a \Rightarrow \mathit{Monoid}\ (\mathit{Set}\ a)\ \textbf{where}$$
$$\quad \mathit{mempty} = \mathit{empty}$$
$$\quad \mathit{mappend} = \mathit{union}$$

Monoid Instances

```
import Data.Map
instance (Ord k) ⇒ Monoid (Map k v) where
  mempty = empty
  mappend = union
```

```
import Text.XHTML
instance Monoid Html
```

```
import Data.Text
instance Monoid Text
```

Monads (Wikipedia survey)

- Monad (Greek philosophy) a term meaning "unit"
- Monad (Biology) a term for simple unicellular organisms
- Monadology, a book of philosophy by Gottfried Leibniz
- Monadologia Physica by Immanuel Kant
- Windows PowerShell, a command line interface for Microsoft Windows code-named "Monad"
- Xmonad, a window manager for the X Window System
- Monad (music), a single note
- Monad, a construction in category theory
- Monad, functional programming constructs that capture various notions of computation

Monads

- Useful abstraction
- Programable semicolon

Monads in Programming Languages

Shall we be pure or impure?
The functional programming community divides into two camps:

- "Pure" languages, such as Haskell, are based directly upon the mathematical notion of a function as a mapping from arguments to results.
- "Impure" languages, such as ML, are based upon the extension of this notion with a range of possible effects, such as exceptions and assignments.

Live Hacking

Abstracting programming patterns

Monads are an example of the idea of abstracting out a common programming pattern as a definition.

$$
\begin{aligned}
&\mathit{inc} :: [\mathit{Int}] \to [\mathit{Int}] \\
&\mathit{inc}\ [\,] \quad\quad\ = [\,] \\
&\mathit{inc}\ (n : ns) = n + 1 : \mathit{inc}\ ns
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{sqr} :: [\mathit{Int}] \to [\mathit{Int}] \\
&\mathit{sqr}\ [\,] \quad\quad\ = [\,] \\
&\mathit{sqr}\ (n : ns) = n \uparrow 2 : \mathit{sqr}\ ns
\end{aligned}
$$

Abstracting programming patterns

Monads are an example of the idea of abstracting out a common programming pattern as a definition.

$$
\begin{aligned}
&map && :: (a \to b) \to [a] \to [b] \\
&map\ f\ [\,] && = [\,] \\
&map\ f\ (x : xs) && = f\ x : map\ f\ xs
\end{aligned}
$$

$$
\begin{aligned}
&inc = map\ (+1) \\
&sqr = map\ (\uparrow 2)
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{class}\ Functor\ f\ \textbf{where} \\
&\quad fmap :: (a \to b) \to f\ a \to f\ b
\end{aligned}
$$

A simple evaluator

$$\textbf{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Div}\ \mathit{Expr}\ \mathit{Expr}$$

$$\begin{aligned}
&\mathit{eval} && :: \mathit{Expr} \rightarrow \mathit{Int} \\
&\mathit{eval}\ (\mathit{Val}\ n) && = n \\
&\mathit{eval}\ (\mathit{Div}\ x\ y) && = \mathit{eval}\ x\ \text{`}\mathit{div}\text{`}\ \mathit{eval}\ y
\end{aligned}$$

A safe evaluator

```
data Maybe a = Nothing | Just a

safediv        :: Int -> Int -> Maybe Int
safediv n m = if m == 0 then Nothing else Just (n `div` m)

eval              :: Expr -> Maybe Int
eval (Val n)      = Just n
eval (Div x y)    = case eval x of
                      Nothing -> Nothing
                      Just n  -> case eval y of
                                   Nothing -> Nothing
                                   Just m  -> safediv n m
```

Combining sequencing and processing

$$
\begin{array}{l}
(\gg\!=) \quad :: \; Maybe\ a \rightarrow (a \rightarrow Maybe\ b) \rightarrow Maybe\ b \\
m \gg\!= f = \mathbf{case}\ m\ \mathbf{of} \\
\qquad\qquad Nothing \rightarrow Nothing \\
\qquad\qquad Just\ x \rightarrow f\ x
\end{array}
$$

Combining sequencing and processing

$$
\begin{aligned}
&(\gg=) && :: \mathit{Maybe}\ a \to (a \to \mathit{Maybe}\ b) \to \mathit{Maybe}\ b \\
&\mathit{Nothing} \gg= _ && = \mathit{Nothing} \\
&(\mathit{Just}\ x) \gg= f && = f\ x
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{eval} && :: \mathit{Expr} \to \mathit{Maybe}\ \mathit{Int} \\
&\mathit{eval}\ (\mathit{Val}\ n) && = \mathit{Just}\ n \\
&\mathit{eval}\ (\mathit{Div}\ x\ y) && = \mathit{eval}\ x \gg= (\lambda n \to \\
& && \quad\ \mathit{eval}\ y \gg= (\lambda m \to \\
& && \quad\ \mathit{safediv}\ n\ m))
\end{aligned}
$$

Combining sequencing and processing

$$
\begin{aligned}
(\ggg\!=) \quad &:: \; \mathit{Maybe}\ a \rightarrow (a \rightarrow \mathit{Maybe}\ b) \rightarrow \mathit{Maybe}\ b \\
\mathit{Nothing} \ggg\!= _ &= \mathit{Nothing} \\
(\mathit{Just}\ x) \ggg\!= f &= f\ x
\end{aligned}
$$

$$
\begin{aligned}
\mathit{eval} \quad &:: \; \mathit{Expr} \rightarrow \mathit{Maybe}\ \mathit{Int} \\
\mathit{eval}\ (\mathit{Val}\ n) &= \mathit{Just}\ n \\
\mathit{eval}\ (\mathit{Div}\ x\ y) &= \mathit{eval}\ x \ggg\!= \lambda n \rightarrow \\
&\quad\; \mathit{eval}\ y \ggg\!= \lambda m \rightarrow \\
&\quad\; \mathit{safediv}\ n\ m
\end{aligned}
$$

Notation

$$
\begin{aligned}
&\mathit{m1} \gg\!= \lambda \mathit{x1} \rightarrow \\
&\mathit{m2} \gg\!= \lambda \mathit{x2} \rightarrow \\
&\ldots \\
&\mathit{mn} \gg\!= \lambda \mathit{xn} \rightarrow \\
&f\ \mathit{x1}\ \mathit{x2}\ \ldots\ \mathit{xn}
\end{aligned}
$$

$$
\begin{aligned}
\textbf{do}\ &\mathit{x1} \leftarrow \mathit{m1} \\
&\mathit{x2} \leftarrow \mathit{m2} \\
&\ldots \\
&\mathit{xn} \leftarrow \mathit{mn} \\
&f\ \mathit{x1}\ \mathit{x2}\ \ldots\ \mathit{xn}
\end{aligned}
$$

Notation

$$
\begin{aligned}
\textbf{do}\ & \mathit{x1} \leftarrow \mathit{m1};\\
& \mathit{x2} \leftarrow \mathit{m2};\\
& \ \ldots\\
& \mathit{xn} \leftarrow \mathit{mn};\\
& f\ \mathit{x1}\ \mathit{x2}\ \ldots\ \mathit{xn}
\end{aligned}
$$

Notation

$$
\begin{aligned}
&m1 \gg\!= \lambda x1 \rightarrow \\
&m2 \gg\!= \lambda x2 \rightarrow \\
&\ldots \\
&mn \gg\!= \lambda xn \rightarrow \\
&f\ x1\ x2\ \ldots\ xn
\end{aligned}
$$

Monads in Haskell

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

$$
\begin{array}{l}
\mathit{return} :: a \to \mathit{Maybe}\ a \\
(\gg\!=)\ :: \mathit{Maybe}\ a \to (a \to \mathit{Maybe}\ b) \to \mathit{Maybe}\ b
\end{array}
$$

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

The list monad

$$
\begin{aligned}
&\mathit{return} :: a \quad \to [a] \\
&(\gg\!=) \; :: [a] \to (a \to [b]) \to [b]
\end{aligned}
$$

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

Monad comprehensions

$$
\begin{aligned}
&\mathit{pairs} :: [a] \to [b] \to [(a, b)] \\
&\mathit{pairs}\ \mathit{xs}\ \mathit{ys} = \mathbf{do}\ x \leftarrow \mathit{xs} \\
&\qquad\qquad\qquad\qquad\ \ y \leftarrow \mathit{ys} \\
&\qquad\qquad\qquad\qquad\ \ \mathit{return}\ (x, y)
\end{aligned}
$$

$$
\mathit{pairs}\ \mathit{xs}\ \mathit{ys} = [(x, y) \mid x \leftarrow \mathit{xs}, y \leftarrow \mathit{ys}]
$$

The state monad

$$\textbf{data}\ State\ s\ a = State\ (s \to (a, s))$$

```
             +------------------+  v :: a
s1 :: s      |                  |----------->
------------>| ma :: State s a  |
             |                  |----------->
             +------------------+  s2 :: s
```

The state monad

$$\textbf{data}\ State\ s\ a = State\ (s \to (a, s))$$

$$f :: Char \to State\ [(Char, Int)]\ Int$$

```
 c :: Char              +---+  i :: Int
----------------------->|   |---------------------->
                        | f |
----------------------->|   |---------------------->
 s1 :: [(Char,Int)]     +---+  s2 :: [(Char,Int)]
```

$$\textbf{data}\ State\ s\ a = State\ (s \rightarrow (a, s))$$

$$\begin{aligned}
&runState :: State\ s\ a \rightarrow s \rightarrow (a, s)\\
&runState\ (State\ f)\ s = f\ s
\end{aligned}$$

$$\begin{aligned}
&return :: a \rightarrow State\ s\ a\\
&(\ggg\!=) :: State\ s\ a \rightarrow (a \rightarrow State\ s\ b) \rightarrow State\ s\ b
\end{aligned}$$

$$\begin{aligned}
&\textbf{instance}\ Monad\ (State\ s)\ \textbf{where}\\
&\quad return\ a\ \ = State\ (\lambda s \rightarrow (a, s))\\
&\quad ma \ggg\!= f = State\ (\lambda s \rightarrow \textbf{let}\ (a, s') = runState\ ma\ s\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \ runState\ (f\ a)\ s')
\end{aligned}$$

Live hacking

- The *State* monad
- The *get* and *put* primitives
- Tree labelling example

The *IO* monad

$$
\begin{aligned}
&\mathit{return} :: a \rightarrow \mathit{IO}\ a \\
&(\gg\!=)\ :: \mathit{IO}\ a \rightarrow (a \rightarrow \mathit{IO}\ b) \rightarrow \mathit{IO}\ b
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{getChar} :: \mathit{IO}\ \mathit{Char} \\
&\mathit{putChar} :: \mathit{Char} \rightarrow \mathit{IO}\ ()
\end{aligned}
$$

The *IO* monad

$$getChar :: IO\ Char$$
$$putChar :: Char \rightarrow IO\ ()$$

$$\begin{aligned}
&getLine :: IO\ String\\
&getLine = \mathbf{do}\ x \leftarrow getChar\\
&\qquad\qquad\quad \mathbf{if}\ x \equiv \texttt{'\textbackslash n'}\\
&\qquad\qquad\qquad \mathbf{then}\ return\ [\,]\\
&\qquad\qquad\qquad \mathbf{else}\ \ \mathbf{do}\ xs \leftarrow getLine\\
&\qquad\qquad\qquad\qquad\qquad return\ (x : xs)
\end{aligned}$$

Conceptual model of *IO*

$$\textbf{type}\ \mathit{RealWorld} = \ldots$$

$$\textbf{type}\ \mathit{IO}\ a = \mathit{State}\ \mathit{RealWorld}\ a$$

$$\textbf{type}\ \mathit{IO}\ a = \mathit{RealWorld} \rightarrow (a, \mathit{RealWorld})$$

Monadic combinators

$$
\begin{aligned}
&\mathit{liftM} && :: \mathit{Monad}\ m \Rightarrow (a \rightarrow b) \rightarrow m\ a \rightarrow m\ b \\
&\mathit{liftM}\ f\ \mathit{mx} && = \mathbf{do}\ x \leftarrow \mathit{mx} \\
& && \qquad\ \ \mathit{return}\ (f\ x)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{join} && :: \mathit{Monad}\ m \Rightarrow m\ (m\ a) \rightarrow m\ a \\
&\mathit{join}\ \mathit{mmx} && = \mathbf{do}\ \mathit{mx} \leftarrow \mathit{mmx} \\
& && \qquad\ \ x \quad\ \leftarrow \mathit{mx} \\
& && \qquad\ \ \mathit{return}\ x
\end{aligned}
$$

Monadic combinators

$$
\begin{aligned}
&(\gg) && :: \mathit{Monad}\ m \Rightarrow m\ a \rightarrow m\ b \rightarrow m\ b \\
&mx \gg my && = \textbf{do}\ _ \leftarrow mx \\
& && \qquad\ \ y \leftarrow my \\
& && \qquad\ \ \mathit{return}\ y
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{sequence} && :: \mathit{Monad}\ m \Rightarrow [m\ a] \rightarrow m\ [a] \\
&\mathit{sequence}\ [\,] && = \mathit{return}\ [\,] \\
&\mathit{sequence}\ (mx : mxs) && = \textbf{do}\ x \ \ \leftarrow mx \\
& && \qquad\ \ xs \leftarrow \mathit{sequence}\ mxs \\
& && \qquad\ \ \mathit{return}\ (x : xs)
\end{aligned}
$$

The monad laws

Left identity:

$$return\ x \gg\!= f = f\ x$$

Right identity:

$$mx \gg\!= return = mx$$

Associativity:

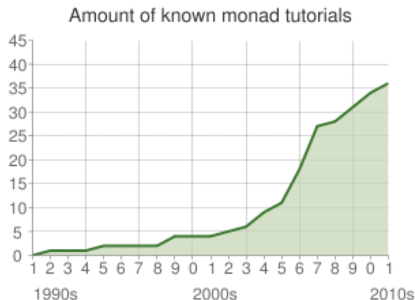
$$(mx \gg\!= f) \gg\!= g = mx \gg\!= (\lambda x \rightarrow (f\ x \gg\!= g))$$

$liftM\ (f \circ g) = liftM\ f \circ liftM\ g$

$$
\begin{aligned}
& (liftM\ f \circ liftM\ g)\ mx \\
= & \quad \text{-- applying .} \\
& liftM\ f\ (liftM\ g\ mx) \\
= & \quad \text{-- applying the second liftM} \\
& liftM\ f\ (mx \ggg\!= \lambda x \rightarrow return\ (g\ x)) \\
= & \quad \text{-- applying liftM} \\
& (mx \ggg\!= \lambda x \rightarrow return\ (g\ x)) \ggg\!= \lambda y \rightarrow return\ (f\ y) \\
= & \quad \text{-- applying the associativity law} \\
& mx \ggg\!= (\lambda z \rightarrow (return\ (g\ z) \ggg\!= \lambda y \rightarrow return\ (f\ y))) \\
= & \quad \text{-- applying the left identity law} \\
& mx \ggg\!= (\lambda z \rightarrow return\ (f\ (g\ z))) \\
= & \quad \text{-- unapplying .} \\
& mx \ggg\!= (\lambda z \rightarrow return\ ((f \circ g)\ z))) \\
= & \quad \text{-- unapplying liftM} \\
& liftM\ (f \circ g)\ mx
\end{aligned}
$$

Reading

- Learn You a Haskell for Great Good! (Chapter 12)
- Real World Haskell (Chapter 14)
- Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. Simon Peyton Jones. `http://research.microsoft.com/en-us/um/people/simonpj/papers/marktoberdorf/`
- Countless monad tutorials?! `http://www.haskell.org/haskellwiki/Monad_tutorials_timeline`



Next lecture

- More monads (e.g., reader, writer, exception and continuation)
- Combining monads (monad transformers)