

# 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:

$$\text{mappend mempty } x = x$$
$$\text{mappend } x \text{ mempty} = x$$
$$\text{mappend } x (\text{mappend } y \ z) = \text{mappend } (\text{mappend } x \ y) \ z$$

## Monoid Instances

**instance** *Monoid* [a] **where**

*mempty* = []

*mappend* = (++)

**import** *Data.Set*

**instance** *Ord* a  $\Rightarrow$  *Monoid* (*Set* a) **where**

*mempty* = *empty*

*mappend* = *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.

$$inc :: [Int] \rightarrow [Int]$$
$$inc [] = []$$
$$inc (n : ns) = n + 1 : inc ns$$
$$sqr :: [Int] \rightarrow [Int]$$
$$sqr [] = []$$
$$sqr (n : ns) = n \uparrow 2 : sqr ns$$

## Abstracting programming patterns

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

```
map      :: (a → b) → [a] → [b]
map f [] = []
map f (x : xs) = f x : map f xs
```

```
inc = map (+1)
sqr = map (↑2)
```

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

## A simple evaluator

**data**  $Expr = Val Int \mid Div Expr Expr$

$eval \quad \quad \quad :: Expr \rightarrow Int$

$eval (Val n) = n$

$eval (Div x y) = eval x 'div' eval y$



## 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

$(\gg=)$  :: *Maybe a*  $\rightarrow$  (*a*  $\rightarrow$  *Maybe b*)  $\rightarrow$  *Maybe b*

*m*  $\gg=$  *f* = **case** *m* **of**

*Nothing*  $\rightarrow$  *Nothing*

*Just x*  $\rightarrow$  *f x*



## Combining sequencing and processing

$(\gg=)$              $:: \text{Maybe } a \rightarrow (a \rightarrow \text{Maybe } b) \rightarrow \text{Maybe } b$

$\text{Nothing } \gg= \_ = \text{Nothing}$

$(\text{Just } x) \gg= f = f \ x$

$\text{eval}$              $:: \text{Expr} \rightarrow \text{Maybe Int}$

$\text{eval } (\text{Val } n) = \text{Just } n$

$\text{eval } (\text{Div } x \ y) = \text{eval } x \gg= \lambda n \rightarrow$   
 $\text{eval } y \gg= \lambda m \rightarrow$   
 $\text{safediv } n \ m$

## Notation

$m_1 \gg \lambda x_1 \rightarrow$

$m_2 \gg \lambda x_2 \rightarrow$

...

$m_n \gg \lambda x_n \rightarrow$

$f\ x_1\ x_2\ \dots\ x_n$

**do**  $x_1 \leftarrow m_1$

$x_2 \leftarrow m_2$

...

$x_n \leftarrow m_n$

$f\ x_1\ x_2\ \dots\ x_n$

## Notation

```
do  $x_1 \leftarrow m_1;$   
     $x_2 \leftarrow m_2;$   
    ...  
     $x_n \leftarrow m_n;$   
     $f\ x_1\ x_2\ \dots\ x_n$ 
```

## Notation

$m1 \gg \lambda x1 \rightarrow$

$m2 \gg \lambda x2 \rightarrow$

...

$mn \gg \lambda xn \rightarrow$

$f x1 x2 \dots xn$

# Monads in Haskell

**class** *Monad* *m* **where**

*return* :: *a* → *m a*

(*>>=*) :: *m a* → (*a* → *m b*) → *m b*

*return* :: *a* → *Maybe a*

(*>>=*) :: *Maybe a* → (*a* → *Maybe b*) → *Maybe b*

**instance** *Monad Maybe* **where**

*return* *x* = *Just x*

*Nothing* *>>=* \_ = *Nothing*

(*Just x*) *>>=* *f* = *f x*



## The list monad

$return :: a \rightarrow [a]$   
 $(\gg=) :: [a] \rightarrow (a \rightarrow [b]) \rightarrow [b]$

**instance** *Monad* [] **where**  
   $return\ x = [x]$   
   $xs\ \gg= f = concat\ (map\ f\ xs)$

## Monad comprehensions

```
pairs :: [a] → [b] → [(a, b)]  
pairs xs ys = do x ← xs  
                y ← ys  
                return (x, y)
```

```
pairs xs ys = [(x, y) | x ← xs, y ← ys]
```

# The state monad

**data** *State* s a = *State* (s → (a, s))

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

## The state monad

**data** *State* *s* *a* = *State* (*s* → (*a*, *s*))

*f* :: *Char* → *State* [(*Char*, *Int*)] *Int*

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

**data** *State* *s a* = *State* (*s* → (*a*, *s*))

*runState* :: *State* *s a* → *s* → (*a*, *s*)

*runState* (*State* *f*) *s* = *f* *s*

*return* :: *a* → *State* *s a*

(*>>=*) :: *State* *s a* → (*a* → *State* *s b*) → *State* *s b*

**instance** *Monad* (*State* *s*) **where**

*return* *a* = *State* (*λs* → (*a*, *s*))

*ma* *>>=* *f* = *State* (*λs* → **let** (*a*, *s'*) = *runState* *ma* *s*  
**in** *runState* (*f* *a*) *s'*)

## Live hacking

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

## The *IO* monad

*return* ::  $a \rightarrow IO\ a$

$(\gg=)$  ::  $IO\ a \rightarrow (a \rightarrow IO\ b) \rightarrow IO\ b$

*getChar* ::  $IO\ Char$

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

## The *IO* monad

```
getChar :: IO Char  
putChar :: Char → IO ()
```

```
getLine :: IO String  
getLine = do x ← getChar  
           if x ≡ '\n'  
           then return []  
           else do xs ← getLine  
                  return (x : xs)
```



## Conceptual model of *IO*

**type** *RealWorld* = ...

**type** *IO a* = *State RealWorld a*

**type** *IO a* = *RealWorld*  $\rightarrow$  (*a*, *RealWorld*)

## Monadic combinators

*liftM* :: *Monad m*  $\Rightarrow$   $(a \rightarrow b) \rightarrow m\ a \rightarrow m\ b$

*liftM f mx* = **do**  $x \leftarrow mx$   
                  *return (f x)*

*join* :: *Monad m*  $\Rightarrow$   $m\ (m\ a) \rightarrow m\ a$

*join mmx* = **do**  $mx \leftarrow mmx$   
                   $x \leftarrow mx$   
                  *return x*



## The monad laws

Left identity:

$$\text{return } x \gg= f = f \ x$$

Right identity:

$$mx \gg= \text{return} = mx$$

Associativity:

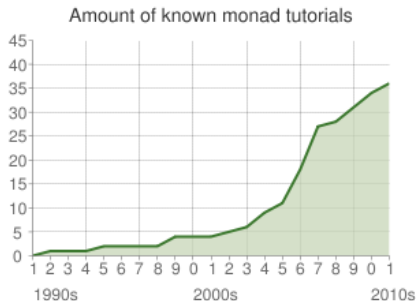
$$(mx \gg= f) \gg= g = mx \gg= (\lambda x \rightarrow (f \ x \gg= g))$$

$$\text{liftM } (f \circ g) = \text{liftM } f \circ \text{liftM } g$$

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

## 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](http://www.haskell.org/haskellwiki/Monad_tutorials_timeline)



## Next lecture

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