

# Programming Languages, CPSC 604—Slides 8

## Monads

Jaakko Järvi

February 24, 2009

# Outline

1 Introduction

2 Mondads

3 Monadic IO

## Definition: first approximation

- Monad is a strategy for combining computations into more complex computations
- No language support, besides higher-order functions, is necessary
  - But provided, e.g., in Haskell

## Example: Maybe

- The `Maybe` data-type can be used for signaling a failure

```
data Maybe a = Just a | Nothing
```

- Suppose we have a partial function `f` of type `a → b`. We can define it with type

```
f :: a → Maybe b
```

to have `Maybe` signal success with wrapping a value to `Just`, and failure with `Nothing`

## Example continues

- Assume the following function querying a database, signaling failure with `Nothing`

```
doQuery :: Query → DB → Maybe Record
```

- Consider the task of performing a sequence of queries.
- Programmer's responsibility to check for failures after every query

```
r :: Maybe Record
```

```
r = case doQuery q1 db of
```

```
  Nothing → Nothing
```

```
  Just r1 → case doQuery (q2 r1) db of
```

```
    Nothing → Nothing
```

```
    Just r2 → case doQuery (q3 r2) db of
```

```
      Nothing → Nothing
```

```
      Just r3 → ...
```

# Capture the pattern into a combinator

```

thenMB :: Maybe a → (a → Maybe b) → Maybe b
mB 'thenMB' f = case mB of
    Nothing → Nothing
    Just a → f a
  
```

- This allows the following rewrite to `doQuery`

```

r :: Maybe Record
r = doQuery q1 db      'thenMB' \r1 →
    doQuery (q2 r1) db 'thenMB' \r2 →
    doQuery (q3 r2) db 'thenMB' ....
  
```

## Another example: state

- Pure functional programming  $\Rightarrow$  no state
- All state must be “threaded through functions”, that is, we must pass a state parameter to a function, and get new state as part of the result
- State type is a function from state to a (result, new state) pair

```
type StateT s a = s  $\rightarrow$  (a, s)
```

- Still on databases... modeling state changing operations. Operations take a record, and a database, and return success/failure and a new database:

```
addRec :: Record  $\rightarrow$  DB  $\rightarrow$  (Bool, DB)
```

```
delRec :: Record  $\rightarrow$  DB  $\rightarrow$  (Bool, DB)
```

- or making it explicit that these functions are state transformers:

```
addRec :: Record  $\rightarrow$  StateT DB Bool
```

```
delRec :: Record  $\rightarrow$  StateT DB Bool
```

## Using the state transformers

- Programming with these directly is again somewhat awkward, requiring one to introduce intermediate variables etc.

```
newDB :: StateT DB Bool
newDB db = let (bool1,db1) = addRec rec1 db
              (bool2,db2) = addRec rec2 db1
              (bool3,db3) = delRec rec3 db2
              in (bool1 && bool2 && bool3,db3)
```

- We can again define a combinator to help, to chain states together, passing result state to the next transformer

```
thenST ::
  StateT s a → (a → StateT s b) → StateT s b
st 'thenST' f = \s → let (v,s') = st s
                      in f v s'
```

- This is handy for making a value into a state transformer:

```
returnST :: a → StateT s a
returnST a = \s → (a,s)
```

# Database updates with the combinator

```
type StateT s a = s → (a, s)
```

```
thenST :: StateT s a → (a → StateT s b) → StateT s b
st 'thenST' f = \s → let (v,s') = st s
                      in f v s'
```

```
returnST :: a → StateT s a
returnST a = \s → (a,s)
```

```
addRec :: Record → StateT DB Bool
delRec :: Record → StateT DB Bool
```

```
newDB :: StateT DB Bool
newDB = addRec rec1 'thenST' \bool1 →
        addRec rec2 'thenST' \bool2 →
        delRec rec3 'thenST' \bool3 →
        returnST (bool1 && bool2 && bool3)
```

# Combinators controlling parameter passing and computational flow

- Many uses for the kind of programming we just saw
  - Data Structures : lists, trees, sets.
  - Computational Flow : Maybe, Error Reporting, non-determinism
  - Value Passing : StateT, environment variables, output generation
  - Interaction with external state: IO, GUI programming, foreign language interfaces
  - Other : parsing combinators, concurrency, mutable data structures.

# Combinators controlling parameter passing and computational flow

- Many uses for the kind of programming we just saw
  - Data Structures : lists, trees, sets.
  - Computational Flow : Maybe, Error Reporting, non-determinism
  - Value Passing : StateT, environment variables, output generation
  - Interaction with external state: IO, GUI programming, foreign language interfaces
  - Other : parsing combinators, concurrency, mutable data structures.
- All the above combinators are instances of *monads*

# Outline

1 Introduction

**2 Mondads**

3 Monadic IO

# Monads and category theory

- Monads arise from category theory, look at works by Eugenio Moggi
- Be prepared for long hours of reading, and a LOT of layers of abstraction
- Introduced to Haskell by Wadler  
Wadler: “The essence of functional programming”, <http://homepages.inf.ed.ac.uk/wadler/papers/essence/essence.ps>

# Definition

- Monad is a triple  $(M, \text{return}, \gg=)$  consisting of a type constructor  $M$  and two polymorphic functions:

$\text{return} :: a \rightarrow M\ a$

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

- which satisfy the *monad laws* (note, checking these is up to the programmer):

$(\text{return}\ a) \gg= f == f\ a$  *-- left unit*

$m \gg= \text{return} == m$  *-- right unit*

$(m \gg= f) \gg= g$   
 $== m \gg= (\backslash x \rightarrow f\ x \gg= g)$  *-- associativity*

- $\gg=$  is called the **bind** operator

# The Monad type class

```
class Monad m where
  >>= :: m a → (a → m b) → m b
  >>  :: m a → m b → m b
  return :: a → m a

  m >> k = m >>= \_ → k
```

- `>>` is just a shorthand for `>>=` ignoring the result of first action
- Any type with compatible combinators can be made to be an instance of this class. For example:

```
data Maybe a = Just a | Nothing
```

```
thenMB :: Maybe a → (a → Maybe b) → Maybe b
```

```
instance Monad Maybe where
  (>>=) = thenMB
  return a = Just a
```

## Utilizing the common notation

- Thanks to common notation, it is possible to define functions operating on *all* monads
- For example, execute one step of each monadic computation in a list

```

sequence      :: Monad m => [m a] -> m [a]
sequence []   = return []
sequence (c:cs) = c          >>= \x ->
                        sequence cs >>= \xs ->
                        return (x:xs)
  
```

# List monad

- Even lists are monads!

```
instance Monad [] where
  m >>= f = concatMap f m
  return x = [x]
```

- where `concatMap` maps a function over a list and concatenates the results:

```
concatMap :: (a -> [b]) -> [a] -> [b]
```

# Reiterating the basic idea

- Converting a program into a monadic form means:
  - A function of type  $a \rightarrow b$  is converted to a function of type  $a \rightarrow M b$
  - $M$  then captures whatever needs to be captured, environment, state, ...
  - and can be changed easily

## Going into, staying in, and getting out?

- Roughly, `return` gets a value into a monad
- `Bind` keeps us in the monad and allows to perform computations within
- There's nothing to get us out!
- For some monads we can define functions that allow us to access the value in the monad, for others no.
- E.g., we can never observe the value in the `IO` (of course, we can manipulate it with `bind`)
- This is crucial for not "leaking" side-effects to otherwise purely functional program
  - Any function whose return type does not include the `IO` type constructor, is guaranteed not to use the `IO` monad (because if the function introduces a use, there is no way to get rid of the type)

# IO in pure languages

- Function like `getChar :: FileHandle → Char` destroys *referential transparency*. One solution:

```
getChar :: FileHandle → World → (Char, World)
```

- Must somehow make sure that `World` can only be used once, otherwise the program needs to store all past `Worlds`
- *Clean* solution: Uniqueness types
- Haskell solution: Monads. Note that `getChar` has the following type, it is a state transformer:

```
getChar :: FileHandle → StateT World Char
```

- The combinators enforce that `World` is always used only once, except it is exposed in the end. The `IO` ADT comes to rescue (`IO` can be made an instance of the `Monad` class):

```
data IO a = IO (StateT World a)
  -- :: World → (a, World)
```

- Now: `getChar :: FileHandle → IO Char`

# IO Examples

- A few IO operations

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

- Monad's functions

```
(>>=) :: IO a → (a → IO b) → IO b
return :: a → IO a
```

- Echo:

```
echo :: IO ()
echo = getChar >>= putChar
```

- Examples from *Peyton-Jones: "Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell"*

# IO Examples ...

- Can we echo twice? `echo >>= echo`

## IO Examples ...

- Can we echo twice? `echo >>= echo`
- No, `>>=` expects a function. This is OK:

```
echo >>= \_ → echo
```

## IO Examples ...

- Can we echo twice? `echo >>= echo`
- No, `>>=` expects a function. This is OK:

```
echo >>= \_ → echo
```

- And as we remember, generalizes to:

```
(>>) :: IO a → IO b → IO b  
(>>) ma mb = ma >>= (\_ → mb)
```

- Allowing:

```
echo >> echo
```

# Do notation

- Consider the following

```
getTwoChars :: IO (Char, Char)
getTwoChars = getChar >>= \c1 →
               getChar >>= \c2 →
               return (c1, c2)
```

- The 'do' notation is syntactic sugar for the above

```
getTwoChars = do { c1 ← getChar ;
                  c2 ← getChar ;
                  return (c1, c2);
                  }
```

## One more example

- Monads really are just a certain way to structure code/types

```
getLine :: IO [Char]
getLine = do { c ← getChar ;
              if c == '\n' then
                return []
              else
                do { cs ← getLine ;
                   return (c:cs)
                 }
            }
```

## And one thought

- Entire Haskell program is wrapped inside the IO monad. Example:

```
main :: IO ()
main = getLine >>= \cs →
      putLine (reverse cs)
```

- We occasionally (hopefully often) branch to non-monadic programs, e.g. `(reverse cs)` here