Monads: Safe Side-effects in Functional Programs

Patrick MacArthur
2011-03-07
Pure Functional Programming

- Biggest advantage: No side effects
  - Easier debugging
  - Lazy evaluation
Pure Functional Programming

- Biggest disadvantage: No side effects
  - Input and output are side effects
  - Some algorithms require global or local state
Applications of Haskell

- GHC: most popular Haskell compiler
- xmonad: tiling window manager
- Darcs: distributed version control system
- List of ~50 companies using Haskell:
  - http://haskell.org/haskellwiki/Haskell_in_industry
  - Examples: Bank of America, Facebook, Google
- Real World Haskell
  - Examples: barcode recognition, JSON parser

Source: O’Sullivan, Bryan, et.al.
Monads

- **Motivation**
  - First use in computer science
  - Early attempts at functional I/O

- **Use in functional I/O**
  - Abstraction
  - Advantages

- **Use in state management**
  - Abstraction
  - Implementation
  - Advantages
Original Motivation

- Original idea presented in (Moggi, 1991)
- Formal reasoning about programs
  - Trivial without side effects
  - Most programs have side-effects
  - Solution: monads/category theory
- Says nothing about implementation
Goal of Functional I/O

- \( f :: \text{Int} \rightarrow \text{Int} \)
  - Cannot perform I/O, access network, access anything in the universe except for its Int argument

- \( f :: \text{Int} \rightarrow \_\_\_\_\_ \text{Int} \)
  - Add something else that gives access to perform I/O
Functional I/O: Dialogue

type Dialogue = [Response] -> [Request]

- Based on lists and recursion
  - Function returns I/O requests
  - Responses given in argument

- Problems
  - hard to synchronize
  - hard to compose multiple I/O actions in one function

Source: Jones and Wadler
Functional I/O: Continuations

main :: Result -> Result

- I/O function
  - Takes continuation function as argument
  - Performs I/O
  - Calls continuation function after completion

- Problem: every function is aware of this model

Source: Jones and Wadler
Monads

- Motivation
  - First use in computer science
  - Early attempts at functional I/O

- Use in functional I/O
  - Abstraction
  - Advantages

- Use in state management
  - Abstraction
  - Implementation
  - Advantages
What is a monad?

- Monads provide a context for actions
  - Examples: `getLine`, `print`
- When performed, an action
  - Possibly uses some side effect
  - Returns value
- Actions may be composed (combined)

Source: Jones and Wadler
Input/Output in Haskell

main :: IO ()
main = do
    name <- getLine
    print $ "Hello, " ++ name
Goal of Functional I/O

- \( f :: \text{Int} \rightarrow \text{Int} \)
  - Cannot perform I/O, access network, access anything in the universe except for its Int argument

- \( f :: \text{Int} \rightarrow \text{IO Int} \)
  - Function in IO monad can perform I/O
Monads: unit/return function

someAction :: IO String
someAction = do
  x <- getline
  return $ "Hello, " ++ x

Source: Jones and Wadler
Monads: bind (\(\gg\gg\gg=\)) function

\[
\text{main :: IO ()}
\]
\[
\text{main = getLine >>= main'}
\]
\[
\text{main' :: String -> IO ()}
\]
\[
\text{main' name = print \(\$ \text{"Hello, " ++ name}\)}
\]

Source: Jones and Wadler
Monad advantages

- Functional in nature
- Built into type system
  - Cannot perform IO action outside an IO function
  - Side effects are controlled
- Opaque
  - Internals of IO hidden from user
- Eliminates boilerplate
  - `Bind` absorb “continuation” code
Monads

- Motivation
  - First use in computer science
  - Early attempts at functional I/O
- Use in functional I/O
  - Abstraction
  - Advantages
- Use in state management
  - Abstraction
  - Implementation
  - Advantages
State monad

- Allows hidden local state management
- Examples:
  - Parser state: remaining text and line number
  - Random number generator: seed

Source: O’Sullivan, Bryan, et.al.
Motivation: Random Numbers

pureRNG :: Int -> Int

-- Make two random numbers
main =
    let x = pureRNG initSeed
        y = pureRNG x
    in print $ show x ++ "," ++ show y

Source: O’Sullivan, Bryan, et.al.
Better Random Numbers

nextRandom :: RNG Int

getTwoRandoms = do
    x <- nextRandom
    y <- nextRandom
    return (x, y)

Source: O’Sullivan, Bryan, et.al.
Better Random Numbers

nextRandom :: RNG Int
twoRandoms :: RNG (Int, Int)

main =
    let (x, y) = evalRNG twoRandoms initSeed
    in print $ show x ++ "," ++ show y

Source: O’Sullivan, Bryan, et.al.
General State: Two issues

- How to transport state
  - Usual monadic functions
  - bind “glue” code
- How to transform state
  - Actions: get and put

Source: O’Sullivan, Bryan, et.al.
State transformation

\[
\text{transform} :: b \rightarrow (s \rightarrow (a, s))
\]

- This function is **curried**
  - Can separate into two functions
  - Separate state transformation from computation

Source: O’Sullivan, Bryan, et.al.
State transportation

type State s a = s -> (a, s)

return x = \s -> (x, s)

m >>= f = \s -> let (x, s') = m s
          in (f x) s'

Source: O’Sullivan, Bryan, et.al.
**Initial State: runState**

- **Input**
  - action $m$
  - initial state $s$

- **Output**
  - final state
  - output data

runState ::
  State $s$ $a$
  -> $s$
  -> $(a, s)$
runState $m$ $s$ = $m$ $s$

Source: O’Sullivan, Bryan, et.al.
State transformation

- **get**: returns state as value
  
  \[
  \text{get :: State } s \rightarrow (s, s)
  \]

- **put**: sets state to passed-in state
  
  \[
  \text{put :: } s \rightarrow \text{State } s ()
  \]

  \[
  \text{put } s = \_ \rightarrow (((), s)}
  \]

Source: O’Sullivan, Bryan, et.al.
Monad advantages

- Built into type system
  - Cannot perform state action outside context
  - Side effects are controlled
- Opaque
  - Internals of state mechanism hidden from user
- Eliminates boilerplate
  - bind absorbs state management code
Conclusion

- Side effects essential to programming
  - I/O is primary example
  - Painful to debug
- Monads provide the solution
  - Provide context for actions with side effects
  - Allow composing actions
  - Side effects cannot escape monadic context
References

“Haskell in Industry.” *The Haskell Programming Language*.  

http://doi.acm.org/10.1145/158511.158524

http://dx.doi.org/10.1016/0890-5401(91)90052-4

Questions?