



Advanced Functional Programming

Week 5 – Monads in General, State Monads

René Thiemann

Department of Computer Science

Monads

Last Week: Functors

- class definition

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```
- structure preserving map-function, `fmap id = id`, `fmap (f . g) = fmap f . fmap g`
- instances: `Maybe`, `[]`, `Either a, () a, (,,) a b`, `Data.Map.Map k`, `Parse`
- not instances: `Data.Set.Set`

Last Week: Development of Parsers for PGM Raw Format

- explicit case-analysis for error-handling and explicit state
- implicit error handling and explicit state: `(>>?)` operation
- implicit error handling and implicit state: `Parse` type and `(==>)` operation

RT (DCS @ UIBK)

Week 5

2/25

Parsing With Implicit Error Handling and Explicit State

```
parseP5_take2 :: L.ByteString -> Maybe (Greymap, L.ByteString)
parseP5_take2 s =
  matchHeader (L8.pack "P5") s      >>?
  skipSpace                        >>?
  getNat                           >>?
  \(width, s1) -> skipSpace s1      >>?
  getNat                           >>?
  \(height, s2) -> skipSpace s2     >>?
  getNat                           >>?
  \(maxGrey, s3) -> getBytes 1 s3   >>?
  (getBytes (width * height) . snd) >>?
  \(bitmap, s) -> Just (Greymap width height maxGrey bitmap, s)
```

observations

- `(>>?)` combines computations
- state `s` is explicitly updated and passed around
- `Just` is used to indicate final result

Parsing With Implicit Error Handling and Implicit State

```
parseRawPGM :: Parse Greymap
parseRawPGM =
    parseWhileWith w2c (not . isSpace) ==> \header -> skipSpaces ==>&
    assert (header == "P5") "invalid raw header" ==>&
    parseNat ==> \width -> skipSpaces ==>&
    parseNat ==> \height -> skipSpaces ==>&
    parseNat ==> \maxGrey ->
    parseByte ==>&
    parseBytes (width * height) ==> \bitmap ->
    identity (Greymap width height maxGrey bitmap)
```

observations

- (`==>`) combines computations
- (`==>&`) is restricted version of (`==>`): `p1 ==>& p2 = p1 ==> \ _ -> p2`
- `identity` turns value into a parsing result

Similarities of Operations

- there is an operator to chain computations
`(>>?) :: Maybe a -> (a -> Maybe b) -> Maybe b`
`(>>=) :: Parse a -> (a -> Parse b) -> Parse b`
known operator to chain I/O actions
`(>>=) :: IO a -> (a -> IO b) -> IO b`
- abstract view: replace concrete type constructors by variable `m`, use (`>>=`) as name of op.
`(>>=) :: m a -> (a -> m b) -> m b`
- there is an operator to lift a plain value into the more complex type without requiring computation
`Just :: a -> Maybe a`
`identity :: a -> Parse a`
`return :: a -> IO a`
- abstract view: replace concrete type constructors by `m` again (use `return` as name)
`return :: a -> m a`

Monads

- identified similarities
 - there is type constructor `m` of kind `* -> *`
 - there is an operation to chain two computations
`(>>=) :: m a -> (a -> m b) -> m b`
the latter computation may depend on the result of the former
 - there is an operator to lift an `a`-element into an `m a`-element
`return :: a -> m a`
- the combination of `m`, `>>=` and `return` form a **monad**
 - `>>=` is called **bind**
 - the three **monad-laws** must be satisfied
 - `return x >>= f = f x` `return` is left neutral
 - `mv >>= return = mv` `return` is right neutral
 - `mv >>= (\x -> f x >>= g) = (mv >>= f) >>= g` `bind` is associative
- examples: `Maybe`, `(>>?)`, `Just` is monad as well as `Parse`, `(>>=)`, `identity`
- another example: `IO`, `(>>=)`, `return` is a monad

Monads in Haskell

- similar to functors, there is a type-class for monads
`class Functor f => Applicative f where ... -- details omitted now`
`class Applicative m => Monad m where`
`(>>=) :: m a -> (a -> m b) -> m b`
`(>>) :: m a -> m b -> m b`
`return :: a -> m a`

`ma >> mb = ma >>= (\ _ -> mb)`
- `IO` and `Maybe` are already instances of `Monad`
- for `Parse`, we will define the instance where
 - `(>>=) = (>>=)`,
 - `(>>) = (>>=&)`, and
 - `return = identity`

Advantages of Abstract Monad Class

- same syntax for all monads: ($\gg=$) and ($\gg$) and `return` and `do-notation`
- write common functions which are available for all monads
 - example 1: ($\gg$)
 - example 2: monadic version of `map`
`mapM :: Monad m => (a -> m b) -> [a] -> m [b]`
`mapM f [] = return []`
`mapM f (x : xs) = f x >>= \ y -> mapM f xs >>= \ ys -> return $ y : ys`
`mapM f (x : xs) = do { y <- f x; ys <- mapM f xs; return $ y : ys }`
 - example 3: monadic version of `map`, ignoring the resulting list
`mapM_ :: Monad m => (a -> m b) -> [a] -> m ()`
`mapM_ f [] = return ()`
`mapM_ f (x : xs) = f x >> mapM_ f xs`
`mapM_ f (x : xs) = do { f x; mapM_ f xs }`
 - example 4: monadic version of `foldl`
`foldM :: Monad m => (b -> a -> m b) -> b -> [a] -> m b`
- simplifies refactoring, e.g., change of monad

Example of Raw PGM Parser, No Do-Notation, Parse-Monad

```
parseRawPGM_noDoNotation :: Parse Greymap
parseRawPGM_noDoNotation =
  parseWhileWith w2c (not . isSpace) >>= \ header ->
    assert (header == "P5") "invalid raw header" >>
      skipSpaces >>
        parseNat >>= \ width ->
          skipSpaces >>
            parseNat >>= \ height ->
              skipSpaces >>
                parseNat >>= \ maxGrey ->
                  parseByte >>
                    parseBytes (width * height) >>= \ bitmap ->
                      return (Greymap width height maxGrey bitmap)
```

Example of Raw PGM Parser, Using Do-Notation and Parse-Monad

```
parseRawPGM :: Parse Greymap
parseRawPGM = do
  header <- parseWhileWith w2c (not . isSpace)
  assert (header == "P5") "invalid raw header"
  skipSpaces
  width <- parseNat
  skipSpaces
  height <- parseNat
  skipSpaces
  maxGrey <- parseNat
  _ <- parseByte
  bitmap <- parseBytes (width * height)
  return (Greymap width height maxGrey bitmap)
```

Example of Raw PGM Parser, Using Do-Notation and Maybe-Monad

```
parseP5_doNotation :: L.ByteString -> Maybe (Greymap, L.ByteString)
parseP5_doNotation s = do
  s <- matchHeader (L8.pack "P5") s
  -- no assert here, part of matchHeader
  s <- skipSpace s
  (width, s) <- getNat s
  s <- skipSpace s
  (height, s) <- getNat s
  s <- skipSpace s
  (maxGrey, s) <- getNat s
  (_, s) <- getBytes 1 s
  (bitmap, s) <- getBytes (width * height) s
  return (Greymap width height maxGrey bitmap, s)
```

Running the Monad

- consider monadic operations
 - chain computations $(\gg=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$
 - lift value $return :: a \rightarrow m\ a$
- operations provide means to enter the monad or stay in the monad m
- question: how to exit the monad, i.e., get return value of type without m
- for most monads, there are dedicated **functions to execute or run the monad**
 - `Parse` $parse :: Parse\ a \rightarrow L.ByteString \rightarrow Either\ String\ a$
 - `Maybe` $maybe :: b \rightarrow (a \rightarrow b) \rightarrow Maybe\ a \rightarrow b$
 - ...
- all these monads can be used to build **pure** functions
- major exception: `IO`
 - no way to get rid of the `IO` in the result type
 - it is not possible to use `IO`-functions within pure functions
- consequence: using a monad is no conflict to staying pure

The State Monad

Monads for Programs using State

- main aim
 - encapsulate some state** that may easily be modified and read throughout the computation
 - states should **not** be explicitly passed around as argument
- idea of monadic computation with state
 - a computation that may read and write to a state of type s ,
 - and returns a result of type a
- example applications
 - state stores set of visited nodes in graph traversal
 - state manages generation of fresh names
 - state manages generation of random numbers
 - write some computed result to the file system
- example monads
 - pure `State` monad
 - `newtype State s a = State { runState :: s -> (a, s) }`
 - simplified and more abstract version of `Parse`: no failure handling
 - `IO` and `ST` provide primitives for stateful computations

Running Example: Quicksort

- quicksort is fast sorting algorithm
- selection of pivot-element can improve or degrade performance
- useful strategy: **random selection of pivot-elements** yields $O(n \cdot \log(n))$ expected runtime
- non-randomized implementation (includes counting comparisons)

```
qsortCount :: Ord a => [a] -> (Integer, [a])
qsortCount [] = (0, [])
qsortCount (x : xs) = let
    (low, high) = partition (< x) xs
    c0 = fromIntegral $ length xs
    (c1, qs1) = qsortCount low
    (c2, qs2) = qsortCount high
in (c0 + c1 + c2, qs1 ++ [x] ++ qs2)
```
- here: pivot is always first element
- consequence: quadratic complexity on sorted input lists

Randomized Quicksort using Monads

- random number generator (rng) is passed as monadic function of type `Int -> m Int`
- required: `rng n` delivers number between 0 and `n`
- main function, without fixing the monad

```
qsortMonadic :: (Monad m, Ord a) => (Int -> m Int) -> [a] -> m (Integer,[a])
qsortMonadic rng = qsortMain where
  qsortMain [] = return $ (0, [])
  qsortMain xs = do
    pos <- rng $ n - 1
    let (xs1, x : xs2) = splitAt pos xs
    let (low, high) = partition (< x) (xs1 ++ xs2)
    let c0 = fromIntegral $ n - 1
    (c1,qs1) <- qsortMain low
    (c2,qs2) <- qsortMain high
    return $ (c0 + c1 + c2, qs1 ++ [x] ++ qs2)
  where n = length xs
```

Randomized Quicksort using IO-Monad

- random numbers can be accessed from the global state (IO can be seen as state monad with a special state: the world)
- implementation in Haskell with $O(n \cdot \log(n))$ complexity, assuming perfect rng

```
-- random number between lower and upper bound
randomRIO :: (Int, Int) -> IO Int
```

```
getRandomIO :: Int -> IO Int
getRandomIO n = randomRIO (0, n)
```

```
qsortIO :: Ord a => [a] -> IO (Integer,[a])
qsortIO = qsortMonadic getRandomIO
```

The Pure State-Monad

```
newtype State s a = State { runState :: s -> (a, s) }
```

```
instance Monad (State s) where
  return a = State ( \ s -> (a, s) )
  st1 >>= aSt2 = State
    (\ s -> let (a, s1) = runState st1 s
              in runState (aSt2 a) s1)
```

```
get :: State s s
get = State ( \ s -> (s, s))
```

```
put :: s -> State s ()
put s = State ( \ _ -> ((), s))
```

- chaining via (`>>=`) very similar to (`==>`) in `Parse`
- two primitives to read and write state: `get` and `put`

Pseudo Random Number Generation

- a pseudo random number generator (rng) is a deterministic algorithm to
 - produce an infinite sequence of numbers
 - which look as if they were really randomly chosen numbers
 - in Haskell: `StdGen` is the type of a pseudo rng
 - `uniformR (a,b) rng` delivers a pair `(r,rng')`
 - `r` is the first random number in the sequence, `r` is between `a` and `b`,
 - `rng'` is the rng that produces the rest of the sequence
 - encapsulate rng in `State`-monad
- ```
type RandomState = State StdGen
```

```
getRandomState :: Int -> RandomState Int
getRandomState n = do
 rng <- get
 let (r,rng') = uniformR (0,n) rng
 put rng'
 return r
```

## Randomized Quicksort as Pure Function

```
qsortStateMain :: Ord a => [a] -> RandomState (Integer, [a])
qsortStateMain = qsortMonadic getRandomState

-- different versions to run the State monad
runState :: State s a -> s -> (a, s)
evalState :: State s a -> s -> a
execState :: State s a -> s -> s

qsortState :: Ord a => Int -> [a] -> (Integer, [a])
qsortState seed xs = evalState (qsortStateMain xs) (mkStdGen seed)

-- mkStdGen generates a pseudo rng from a starting seed value
```

## Store Pseudo-RNG via IO References

```
newIORef :: a -> IO (IORef a) -- initial value
readIORef :: IORef a -> IO a -- get
writeIORef :: IORef a -> a -> IO () -- put

-- deterministic pseudo-rng based quicksort using IO references
qsortIOSeed :: Ord a => Int -> [a] -> IO (Integer, [a])
qsortIOSeed seed xs = do
 rngRef <- newIORef (mkStdGen seed)
 let getRandIO n = do
 rng <- readIORef rngRef
 let (r, rng') = uniformR (0,n) rng
 writeIORef rngRef rng'
 return r
 qsortMonadic getRandIO xs

-- a bit faster than State, but cannot be used in pure functions
```

## Store Pseudo-RNG via ST References

```
newSTRef :: a -> ST s (STRef s a) -- initial value
readSTRef :: STRef s a -> ST s a -- get
writeSTRef :: STRef s a -> a -> ST s () -- put

-- deterministic pseudo-rng based quicksort using ST monad and ST references
qsortMonadicST :: Ord a => Int -> [a] -> ST s (Integer, [a])
qsortMonadicST seed xs = do
 rngRef <- newSTRef (mkStdGen seed)
 let getRandST n = do
 rng <- readSTRef rngRef
 let (r, rng') = uniformR (0,n) rng
 writeSTRef rngRef rng'
 return r
 qsortMonadic getRandST xs

-- similar to previos slide, replace IO by ST s, and IORef by STRef s
```

## The ST Monad

- ST: state threads
- `ST s a` is monadic value with state thread `s` and result `a`
- ST monad provides **mutable state** in module `Data.STRef` and **mutable arrays** in `Data.Array.ST`
- in comparison to IO, there is a way to **escape the ST monad** via `runST :: (forall s. ST s a) -> a`  
-- pure version of quicksort-ST-version  
`qsortSTSeed :: Ord a => Int -> [a] -> (Integer, [a])`  
`qsortSTSeed seed xs = runST $ qsortMonadicST seed xs`
- quantification of `s` prevents that result `a` depends on `s`, so `a` cannot be `STRef s b`  
-- fortunately, does not type-check  
`bad = let ref = runST (do ref <- newSTRef True`  
`writeSTRef ref False`  
`return ref)`  
`in runST (readSTRef ref)`
- in total: ST permits us to write **pure Haskell code using mutable state**

## Literature

- Real World Haskell, Chapter 14
- John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. PLDI '94, pages 24 – 35, <https://doi.org/10.1145/178243.178246>