Table of Contents

Common Monads

Benson Joeris
http://bjoeris.com

pluralsight
hardcore developer training

Overview

- Reader
- State
- ST

Table of Contents

Reader Monad

```
import Control.Monad.Reader
```

```
data Reader r a
```

```
instance Monad (Reader r)
```

```
ask :: Reader r r
```

```
runReader :: Reader r a -> r -> a
```

Reader Monad

```
getFirst :: Reader String String
getFirst = do
  name <- ask
  return (name ++ " woke up")
getSecond :: Reader String String
getSecond = do
  name <- ask
  return (name ++ " wrote some Haskell")
getStory :: Reader String String
getStory = do
  first <- getFirst
  second <- getSecond
  return ("First, " ++ first ++
          ". Second, " ++ second ++ ".")
story = runReader getStory "Benson"
```

Reader Monad

```
GHCi> story
Result: "First, Benson woke up.  Second, Benson wrote
some Haskell."
```

Table of Contents

State Monad

```
import Control.Monad.State
```

```
data State s a
```

```
instance Monad (State s)
```

State Monad

```
get :: State s s
```

```
put :: s -> State s ()
```

```
evalState :: State s a -> s -> a
```

State Monad

```
harmonicStep :: State (Double,Double) Double
harmonicStep = do
  (position,velocity) <- get
  let acceleration = (-0.01 * position)
      velocity'    = velocity + acceleration
      position'    = position + velocity'
  put (position',velocity')
  return position
```

```
harmonic :: State (Double,Double) [Double]
harmonic = do
  position <- harmonicStep
  laterPositions <- harmonic
  return (position : laterPositions)
```

State Monad

```
harmonicStep :: State (Double,Double) Double
harmonic     :: State (Double,Double) [Double]
```

```
GHCi> let positions = evalState harmonic (1,0)
GHCi> take 8 positions
Result: [1.0,
         0.99,
         0.9701,
         0.940499,
         0.9014930099999999,
         0.8534720898999999,
         0.7969164489009999,
         0.7323916434129899]
```

State Monad

```
newtype State s a = State (s -> (a, s))
```

Table of Contents

ST Monad

- Implement imperative algorithms
- Modifiable values
- Pure from the outside

ST Monad

```
import Control.Monad.ST
```

```
data ST s a
```

```
instance Monad (ST s)
```

```
runST :: ST s a -> a
```

ST Monad

```
import Data.STRef
```

```
data STRef s a
```

```
newSTRef :: a -> ST s (STRef s a)
```

```
readSTRef :: STRef s a -> ST s a
```

```
writeSTRef :: STRef s a -> a -> ST s ()
```

ST Monad

```
sumST :: [Int] -> STRef s Int -> ST s ()
sumST []       accumRef = return ()
sumST (x : xs) accumRef = do
  accum <- readSTRef accumRef
  writeSTRef accumRef (x + accum)
  sumST xs accumRef
```

```
sum' :: [Int] -> Int
sum' xs = runST $ do
  accumRef <- newSTRef 0
  sumST xs accumRef
  readSTRef accumRef
```

ST Monad Uses

- High performance
- Translating imperative code
- Complicated, multi-part state

Table of Contents

Summary

- Reader
- State
- ST