# An Introduction to Monads

Phillip Mates

March 6, 2012

# Why Monads?

In a purely functional language:

- How do you encode actions with side-effects, such as reading and writing files?
- Is there an elegant way to pass around program state without explicitly threading it in and out of every function?
- ► How do you code up doubly nested for-loops?
- ► What about: Continuation passing style, Writing logs, Memory transactions...

## What are Monads?

They're a very general abstraction idea that can be thought of as:

- containers that wrap values and are composable
- ► the inverse of pointers
- ► an abstraction for modeling sequential actions
- ▶ ...

## Error handling with Maybe

```
-- Does Pony's friend have a friend in animalMap?
animalFriendLookup :: [(String, String)] -> Maybe String
animalFriendLookup animalMap =
  case lookup "Pony" animalMap of
       Nothing -> Nothing
       Just ponyFriend ->
         case lookup ponyFriend animalMap of
              Nothing -> Nothing
              Just ponyFriendFriend ->
                case lookup ponyFriendFriend animalMap of
                     Nothing -> Nothing
                     Just friend -> Just friend
```

#### Monads are comprised of two functions

-- Bind (>>=) :: m a -> (a -> m b) -> m b -- Inject value into a container return :: a -> m a

## Maybe Monad

-- return :: a -> m a return x = Just x

## Using Maybe as a Monad

monadicFriendLookup :: [(String, String)] -> Maybe String
monadicFriendLookup animalMap =

lookup "Pony" animalMap >>= (\ponyFriend -> lookup ponyFriend animalMap >>= (\pony2ndFriend -> lookup pony2ndFriend animalMap >>= (\friend -> Just friend)))

## Using Maybe as a Monad

```
-- or even better:

sugaryFriendLookup :: [(String, String)] -> Maybe String
sugaryFriendLookup animalMap = do
ponyFriend <- lookup "Pony" animalMap
ponyFriend' <- lookup ponyFriend animalMap
ponyFriend'' <- lookup ponyFriend' animalMap
return friend</pre>
```

## Threading program state

```
type Sexpr = String
```

-- naive generation of unique symbol transformStmt :: Sexpr -> Int -> (Sexpr, Int) transformStmt expr counter = (newExpr, counter+1) where newExpr = "(define " ++ var ++ " " ++ expr ++ ")" var = "tmpVar" ++ (show counter)

## Generalizing the threading of state

```
Let's drop
Int -> (Sexpr, Int)
from
transformStmt :: Sexpr -> Int -> (Sexpr, Int)
and replace it with a more general type constructor:
```

#### Generalizing the threading of state

```
Let's drop
Int -> (Sexpr, Int)
from
transformStmt :: Sexpr -> Int -> (Sexpr, Int)
and replace it with a more general type constructor:
```

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

#### State Monad

-- return :: a -> State s a return a = State (\s -> (a, s))

-- (>>=) :: State s a -> (a -> State s b) -> State s b m >>= k = State (\s -> let (a, s') = runState m s in runState (k a) s')

## State Monad Example

#### What can be a Monad?

```
Type constructors with an arity of one, for instance:
-- this can't because it has arity 2:
ghci> :kind State
* -> * -> *
```

```
-- but these have arity 1:
ghci> :kind (State Int)
* -> *
```

```
ghci> :kind []
* -> *
```

#### Deriving the list monad

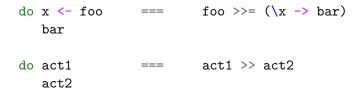
```
ghci> :type (>>=)
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
ghci> :type map
map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
ghci> :type flip map
flip map :: [a] -> (a -> b) -> [b]
ghci> :type concat
concat :: [[a]] -> [a]
```

## The List monad models non-determinism

return x = [x]
xs >>= f = concat (map f xs)

#### The List monad models non-determinism

# Desugaring do Blocks



# Further Topics & Reading

- Monad Transformers
- ► "Real World Haskell" by O'Sullivan, Stewart, and Goerzen
- Corresponding blog post: quined.net/articles/monads.html