LECTURE 12

Theory and Design of PL (CS 538)

March 2, 2020

NEWS

MIDTERM

- **This Friday from 2:30-3:45 in CS 1325**
  - Makeup: **Wednesday from 6:00-7:15 in TBA**
- Exam format
  - 75-minutes, short-answer, in-class exam
  - No aids/notes/electronics allowed
- How to prepare
  - Written (~25%): look at WRs, solutions
  - Programming (~75%): read/write Haskell (HW3)

COURSE EVALS

- Midterm course evals open now until Friday
- Please put feedback on anything in this course
- I will read and think about all of your feedback

APPLICATIVE VERSUS MONADIC

REVIEW: APPLICATIVE AND MONAD

```
class Functor f => Applicative f where
  pure    :: a -> f a                      -- Required op. 1: pure
  (<*>)   :: f (a -> b) -> f a -> f b -- Required op. 2: ap

class Applicative m => Monad m where
  return :: a -> m a                       -- Required op. 1: return
  (>>=)   :: m a -> (a -> m b) -> m b -- Required op. 2: bind
```

REMEMBER: CALCULATOR

- Grammar of a simple calculator language

```
term = atom "+" term | atom "-" term | atom ;
atom = num | "(" term ")" ;
```

- Model with these two Haskell datatypes:

```haskell
data Term = Add Atom Term | Sub Atom Term | Single Atom
data Atom = Num Int | Parens Term
```

APPLICATIVE-STYLE PARSING

- Use applicative/alternative instances for Parser:

```
termP :: Parser Term
termP =  Add <$> atomP <* (tokenP (charP '+')) <*> termP
     <|> Sub <$> atomP <* (tokenP (charP '-')) <*> termP
     <|> Single <$> atomP

atomP :: Parser Atom
atomP =  Num <$> intP
     <|> Parens <$>
         (tokenP $ charP '(') *> termP <* (tokenP $ charP ')')
```

PARSER HAS A MONAD INSTANCE

- But Parser also has a Monad instance:

```
termP' =  do a <- atomP'
             tokenP (charP '+')
             t <- termP'
             return $ Add a t
      <|> do a <- atomP'
             tokenP (charP '-')
             t <- termP'
             return $ Sub a t
      <|> do a <- atomP'
             return $ Single a   -- or: Single <$> a

atomP' =  do i <- intP
             return $ Num i
      <|> -- ...
```

COMPARING THE TWO STYLES

- The two styles can be freely mixed
- Applicative-style
  - Shorter, more condensed
  - Can be complicated to ignore/keep things
- Monadic-style (do-notation)
  - Longer, more verbose
  - Sequential steps are clearer (imperative)

MONADS AS COMPUTATIONS

WHAT "IS" A MONAD?

- No single interpretation—it's just a pattern!
- Useful for describing "computations"
- Take a type `m a`
  - `a` is the type of stuff that is "returned"
  - `m` augments `a` with "side information"
- `m a` is a type of *computation* returning stuff of type `a`

"PROGRAMMABLE SEMICOLON"

- Usually: function composition for sequencing
- But: sometimes we want fancier behavior
  - Maintain a log, update state, …
- But: sometimes we don't want to execute just yet
  - Store for processing later, add more steps, …
- Monad instances let us define how to sequence stuff
  - Handle "plumbing" for side information

ASSEMBLE A COMPUTATION WITHOUT RUNNING IT

- Distinguish between *normal values* and *computations*
  - Cakes versus recipes
- Use (Haskell) programs to build (monadic) programs
  - Pass computations around
  - Repeat computations
  - Combine computations in custom ways
- Only run computations *when* and *where* we want

SIDE-EFFECTS AND PURITY

WHAT IS A SIDE-EFFECT?

- Anything a function depends on besides input
  - Reading a configuration file
  - Getting the current local time
- Anything a function does besides making output
  - Establishing a network connection
  - Opening the pod bay doors

PURITY: NO SIDE-EFFECTS

- All functions in Haskell are pure
- Use monads to express side-effecting computations
  - Need to specifically indicate in types
  - Note: not all monads model side-effects

SIMPLER TO THINK ABOUT

- Function output depends only on the input
  - No hidden state
  - No hidden dependencies
  - No hidden actions
- Input is a lot simpler than state of the world

EASIER TO TEST

- Doesn't depend on external environment
- Totally repeatable and deterministic
  - If it does X, it will *always* do X
  - No matter time of day, other parts of program, etc.

WON'T INDIRECTLY AFFECT OTHER COMPONENTS

- Code changes will only affect input-output
  - Won't step on some shared state
  - Won't mess up other components "indirectly"
- *Modularity* and *abstraction* taken to the limit
  - Callers can only observe input/output
  - Can never be affected by calling a function

WHAT IF WE NEED SIDE-EFFECTS?

SOMETIMES, HAVE TO...

- Want the program to *do something* when we run it!
  - Print to the screen
  - Write a file
  - Turn on the lights
- Would be bad if we could never do this

EFFECTS MARKED IN TYPES

- Haskell effects managed by monads
- Types say: danger!
  - May do stuff besides returning a value
  - May modify hidden state, do I/O, ...
- Allowed effects depend on the kind of monad

REVISITING THE STATE MONAD

STATEFUL PROGRAMS

- "Function with state" produces output, *and* transforms state
- A few ingredients:
  - State is of type `s`
  - Output is of type `a`
- Take start state to output value, plus new state

```
data State s a = MkState (s -> (a, s))
```

MAKING IT INTO A MONAD

- Return: turn ordinary output value into stateful program producing that value
- Bind is a bit more complicated
  1. Run first stateful program
  2. Look at the output value of the first part
  3. Select and run second stateful program

MORE CONCRETELY...

```haskell
return :: a -> State s a
return val = MkState (\state -> (val, state))
--                        ^-State unchanged-^
```

```haskell
(>>=) :: State s a -> (a -> State s b) -> State s b
first >>= f = MkState $ \st ->
  case first of
    MkState stTrans1 -> let (out', st') = stTrans1 st
--                                                      ^--- Run part 1
      in case (f out') of MkState stTrans2 -> stTrans2 st'
--                ^--- Select part 2                        ^--- Run part 2
```

GETTING THE RESULT OUT

- Given stateful program, how do we “run” it?
  - How do we get the result out?
- Need: initial value of the state
- Running turns `State s a` into a normal value

```
runState :: State s a -> s -> (a, s)
runState (MkState stateTrans) initState = stateTrans initState
```

A MONAD FOR ARBITRARY SIDE-EFFECTS

HASKELL MAIN PROGRAM

```
main :: IO ()
main = ...
```

- Why does `main` always have this type in Haskell?
- What is this type, really?
  - In fact, `IO` is a monad!
  - `()` is the “return” type

IO IS A SPECIAL MONAD

- All side-effects are allowed!
  - Can do general input/output actions
  - Can interact with the external world
- This is the *only* place where input/output allowed
- No Haskell definition—it is a completely built-in type

GETTING THE RESULT OUT (?)

- Say we have a `IO Int`. How to get the Int out?
- Is there a function of type `IO Int -> Int`?

There is no way to do this!

- Why? `IO Int` gives Int *and may do real-world stuff*
  - Can't turn this into a pure computation

PROGRAMMING WITH IO

CONSOLE OUTPUT

- All basic printing functions “live in IO”

```
putChar  :: Char   -> IO ()   -- Print a character
putStr   :: String -> IO ()   -- Print a string
putStrLn :: String -> IO ()   -- Print a string and newline
```

```
main :: IO ()
main = do
  putChar  "Q"
  putStr   " is my favorite character.\n"
  putStrLn "Tada!"
```

CONSOLE INPUT

- All basic reading functions “live in IO”

```
getChar  :: IO Char     -- Read a character from console
getLine  :: IO String   -- Read a string from console
```

```
main :: IO ()
main = do
  putStrLn "Enter a character: "
  c   <- getChar
  putStrLn "\nEnter a string: "
  str <- getLine
  putStrLn $ "Got: " ++ c ++ " and " ++ str
```

INTERACT

- Useful utility: read a string, transform it, print it

```
interact :: (String -> String) -> IO () -- Read, transform, print
```

- Pattern: separate pure and impure functions

```
-- Very complicated processing, but pure function
complicatedPureStuff :: String -> String
complicatedPureStuff str = ...

main :: IO ()
main = do
  putStr "Enter something! "
  interact complicatedPureStuff
  putStrLn "\n All done!"
```

FILESYSTEM I/O

- Type of file system paths (depends on system)

```
type FilePath = ...
```

- Library functions for reading/writing: all in I/O!

```
-- Read file into a string
readFile   :: FilePath -> IO String

-- Write string to file
writeFile  :: FilePath -> String -> IO ()

-- Append string to file
appendFile :: FilePath -> String -> IO ()
```

MUTABLE REFERENCES

- In pure Haskell, variables can't be changed/mutated
- In IO monad, can make *mutable references*

```
data IORef a                            -- Built-in type

newIORef   :: a -> IO (IORef a)         -- New cell w/initial value
readIORef  :: IORef a -> IO a           -- Read contents from cell
writeIORef :: a -> IORef a -> IO ()     -- Write contents to cell
```

IMPERATIVE HASKELL

- Operations in IO allow imperative programming

```haskell
main :: IO ()
main = do
  myRef <- newIORef 0                                    -- New counter, init 0
  count <- readIORef myRef                               -- Read count
  putStrLn $ "Count: " ++ (show count)
  writeIORef (count + 1) myRef                           -- Update count
  count' <- readIORef myRef                              -- Read again
  putStrLn $ "Count: " ++ (show count')
```

- Quite a lot of trouble just to increment a counter…
  - Only use where absolutely necessary
  - Pure functions strongly preferred in Haskell

ORGANIZING I/O IN HASKELL

THINKING ABOUT IO

"WORLD TRANSFORMER"

- Idea: running `IO a` can literally change the world
- Imagine IO as the biggest State monad ever:

```
data State s a = MkState (s          -> (a, s         ))
data IO      a = MkIO    (WorldState -> (a, WorldState))
-- Think: IO a === State WorldState a
```

- `WorldState` is the state of the *whole world*
  - Note: this isn't actually how it works

WHEN DO IO EFFECTS “HAPPEN”?

- No matter what, all Haskell functions are pure
- Suppose: call a function of type `IO a -> IO b`
  - Just *changes* one computation into another
  - This *doesn’t* cause effects!

IO EFFECTS HAPPEN "EXTERNALLY"

- Side effects actually take place when `IO a` is "run"
  - But: can't run IO directly within Haskell!
- One way to think about IO in Haskell
  1. Build up a huge side-effecting computation (`main`)
  2. Hand whole thing off for user to run

"CAKE RECIPE"

- If something has type `Cake`, it's a *cake*
- If something has type `IO Cake`, it's a *cake recipe*

> *A cake recipe is different from a cake!*

BUILDING RECIPES

- Monad operations: combine recipes together
  - Do recipes one after another
  - Add extra steps to a recipe
  - Choose between different recipes

> *But no matter what, we will always have just a cake recipe, and not a cake!*

HOW DO WE GET THE CAKE?

- The whole recipe: special symbol `main`, type `IO ()`
- Purpose of Haskell programs is to build this recipe
- The actual cake is made when program is *executed*

PROGRAMMING WITH MONADS

SEQUENCING

- Given: list of computations each returning `a`
- Build: computation returning list of `a`

```
sequence :: Monad m => [m a] -> m [a]
sequence []           = return []
sequence (comp:comps) = do res  <- comp
                           rest <- sequence comps
                           return (res:rest)
```

REPEATING

- Given: integer count and computation returning `a`
- Build: computation returning list of `a`

```
replicateM :: Monad m => Int -> m a -> m [a]
replicateM 0 comp = return []
replicateM n comp = do res  <- comp
                       res' <- replicateM (n - 1) comp
                       return (res:res')
```

MAPPING

- Given:
  - Function from `a` to computation returning `b`
  - A list of `a`'s
- Build:
  - Computation returning list of `b`'s

```haskell
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f []      = return []
mapM f (x:xs) = do res  <- f x
                   res' <- mapM f xs
                   return (res:res')
```

FOLDING

- Given:
  - Function: `b` and `a` to computation returning `b`
  - Initial value of accumulator `b`
  - List of `a`’s
- Build:
  - Computation “folding” list into a `b`

```
foldM :: Monad m => (b -> a -> m b) -> b -> [a] -> m b
foldM f seed []     = return seed
foldM f seed (x:xs) = do accum <- foldM f seed xs
                         f accum x
```