Functional Programming

Input/Output

H. Turgut Uyar

2013-2017

License

Topics

I/O Model

- how can I/O fit into the functional model?
- how about a function that reads in a value of the desired type from the input?
  ```
  inputInt :: Integer
  ```
- breaks reasoning:
  ```
  inputDiff = inputInt - inputInt
  ```
- any function might be affected:
  ```
  foo :: Integer -> Integer
  foo n = inputInt + n
  ```

I/O Type

- new type: `IO a`
  a program which will do some I/O and return a value of type a
- instead of:
  `inputInt :: Integer`
- we have:
  `inputInt :: IO Integer`
- no longer valid:
  `inputInt - inputInt`
  `inputInt + n`

I/O Type

- if I/O doesn't produce a result: `IO ()`
- output:
  `putStr :: String -> IO ()`
  `putStrLn :: String -> IO ()`
- input:
  `getLine :: IO String`

Program Start

- entry point of the program: `main`

example: Hello, world!

```
main :: IO ()
main = putStrLn "Hello, world!"
```


String Conversions

- convert a type to string: `show`
- convert a string to another type: `read`

examples

```
show 42   ~> "42"
show 3.14 ~> "3.14"

read "42" :: Integer ~> 42
read "42" :: Float   ~> 42.0

read "3.14" :: Float ~> 3.14
```

Action Sequences

- I/O consists of actions happening in a sequence
- create an action sequence: do
- small imperative programming language

```
do action1
   action2
   ...
```


Sequence Example

print a string 4 times

```
put4times :: String -> IO ()
put4times str = do putStrLn str
                   putStrLn str
                   putStrLn str
                   putStrLn str
```


Capturing Values

- capture value produced by the program: <-
- can only be used within the sequence

example: reverse and print the line read from the input

```
reverseLine :: IO ()
reverseLine = do line <- getLine
                 putStrLn (reverse line)
```


Local Definitions

- local definitions: let
- can only be used within the sequence

example: reverse two lines

```
reverse2lines :: IO ()
reverse2lines = do line1 <- getLine
                   line2 <- getLine
                   let rev1 = reverse line1
                   let rev2 = reverse line2
                   putStrLn rev2
                   putStrLn rev1
```

Returning Values

- returning result of sequence: `return`

example: read an integer from the input

```
getInteger :: IO Integer
getInteger = do line <- getLine
                return (read line :: Integer)
```


Recursion in Sequence

copy input to output indefinitely

```
copy :: IO ()
copy = do line <- getLine
          putStrLn line
          copy
```


Conditional in Sequence

copy input to output a number of times

```
copyN :: Integer -> IO ()
copyN n = if n <= 0
          then return ()
          else do line <- getLine
                  putStrLn line
                  copyN (n - 1)
```


Conditional in Sequence

copy until input line is empty

```
copyUntilEmpty :: IO ()
copyUntilEmpty = do line <- getLine
                    if line == ""
                       then return ()
                       else do putStrLn line
                               copyUntilEmpty
```

Rock - Paper - Scissors

- two players repeatedly play Rock-Paper-Scissors

data types

```
data Move = Rock | Paper | Scissors
            deriving Show

type Match = ([Move], [Move])

-- moves in reverse order
-- ex: ([Rock, Rock, Paper], [Scissors, Paper, Rock])
```


Outcome

outcome of one round

- A wins $\mapsto$ 1, B wins $\mapsto$ -1, tie $\mapsto$ 0

```
outcome :: Move -> Move -> Integer
outcome mA mB = case (mA, mB) of
    (Rock,     Scissors) ->  1
    (Scissors, Rock)     -> -1
    (Paper,    Rock)     ->  1
    (Rock,     Paper)    -> -1
    (Scissors, Paper)    ->  1
    (Paper,    Scissors) -> -1
    _                    ->  0
```

- exercise: determine the outcome of a match
  `matchOutcome ([Rock, Paper], [Paper, Scissors]) ~> -2`

String Conversions

convert a round in the game to string

```
showRound :: Move -> Move -> String
showRound mA mB = "A plays: " ++ show mA
    ++ ", B plays: " ++ show mB
```

- exercise: convert match result to string

```
showResult ([Rock, Paper], [Paper, Scissors])
   ~> "Player B wins by 2"
showResult ([Rock, Paper], [Paper, Rock])
   ~> "It's a tie"
```


Strategies

- strategy: selects move based on previous moves of opponent
  `[Move] -> Move`

always play the same move

```
rock, paper, scissors :: [Move] -> Move
rock     _ = Rock
paper    _ = Paper
scissors _ = Scissors
```

Strategies

cycle through the options

```
cycle :: [Move] -> Move
cycle ms = case (length ms) `mod` 3 of
    0 -> Rock
    1 -> Paper
    2 -> Scissors
```


Strategies

play whatever opponent played last

```
echo :: [Move] -> Move
echo []          = Rock
echo (latest:_) = latest
```


Interactive Play

- player A: human
- player B: computer, plays echo

convert a character into a move

```
convertMove :: Char -> Move
convertMove c
  | c `elem` "rR" = Rock
  | c `elem` "pP" = Paper
  | c `elem` "sS" = Scissors
  | otherwise     = error "unknown move"
```


Game Play

```
playRound :: Match -> IO ()
playRound match@(movesA, movesB) = do
    ch <- getChar
    putStrLn ""
    if ch == '.'
       then putStrLn (showResult match)
       else do let moveA = convertMove ch
               let moveB = echo movesA
               putStrLn (showRound moveA moveB)
               playRound (moveA : movesA, moveB : movesB)

playInteractive :: IO ()
playInteractive = playRound ([], [])
```

Automatic Play

generate match: `cycle` versus `echo`

```
generateMatch :: Integer -> Match
generateMatch 0 = ([], [])
generateMatch n = step (generateMatch (n - 1))
  where
    step :: Match -> Match
    step (movesA, movesB) =
        (cycle movesB : movesA, echo movesA : movesB)
```


References

Required Reading: Thompson

- Chapter 8: Playing the game: I/O in Haskell