universität innsbruck

WS 2019/2020

Functional Programming

Part 8 – Input and Output

René Thiemann Benedikt Dornauer Maximilian W. Haslbeck
Bart Keulen Florian Meßner Jonas Schöpf
Vincent van Oostrom Xiang Zhang

Department of Computer Science

I/O: Input and Output

- aim: communicate with the user
  - ask user for inputs
  - print answers
  - outside the GHCI read-eval-print-loop
  - stand-alone programs that neither require ghc-installation nor Haskell knowledge of user
- I/O is not restricted to text-based user-I/O
  - reading and writing of files
    (e.g., compiler translates .hs to .exe, or .tex to .pdf)
  - reading and writing into memory
    (mutable state, arrays)
  - reading and writing of network channels
    (e.g., web-server and internet-browser)
  - start other programs and communicate with them
  - play/record sound, mouse-movements, ...

An Initial Example

- ```
  -- file: welcomeIO.hs
  main = do
    putStrLn "Greetings! Please tell me your name."
    name <- getLine
    putStrLn $
      "Welcome to Haskell's IO, " ++ name ++ "!"
  ```
- compile it with GHC (not GHCI) via
  ```
  $ ghc --make welcomeIO.hs
  ```
- and run it
  ```
  $ ./welcomeIO              # welcomeIO.exe on Windows
  Greetings! Please tell me your name.
  Homer                      # this was typed in
  Welcome to Haskell's IO, Homer!
  ```
- notes
  - `putStrLn` – prints string followed by newline
  - `getLine` – reads line from standard input
  - new syntax: `do` and `<-`
  ```

I/O and the Type System

- consider
```
ghci> :l welcomeIO.hs
ghci> :t putStrLn
putStrLn :: String -> IO ()
ghci> :t getLine
getLine :: IO String
ghci> :t main
main :: IO ()
```
- `IO a` is type of I/O actions delivering results of type `a` (in addition to their I/O operations)
- examples
  - `String -> IO ()` – after supplying a string, we obtain an I/O action (in case of `putStrLn`, "printing")
  - `IO ()` – just I/O (in case of `main`, run our program)
  - `IO String` – do some I/O and deliver a string (in case of `getLine`, user-input)

Combining I/O Actions

- I/O actions can be combined
- core building block: bind (syntax >>=)

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

- consider `act_1 >>= \ x -> act_2`
  - on evaluation, this expressions first performs action `act_1`
  - the result of action `act_1` is stored in `x`
  - afterwards action `act_2` is performed (which may depend on `x`)
  - in total, both actions are performed and the result is that of `act_2`
- example

```
putStrLn "hello" >>=
\ _ -> getLine >>=
\ name -> let answer = "hi " ++ name ++ "!" in
putStrLn answer
```

  - result of first `putStrLn` is ignored
  - the answer in `let`-expression is computed purely functional
  - the type is `IO ()`, that of the last I/O action `putStrLn answer`
  - execution of actions if sequential, like in imperative programming

Do-Notation

- there is special syntax for combinations of binds, lambdas and lets

```
do x <- act              =     act >>= \ x -> do block
   block

do act                   =     act >>= \ _ -> do block
   block

do let x = e             =     let x = e in do block
   block
```

- `putStrLn "hello" >>=`
  `\ _ -> getLine >>=`
  `\ name -> let answer = "hi " ++ name ++ "." in`
  `putStrLn answer`
  can be written as

```
do putStrLn "hello"
   name <- getLine
   let answer = "hi " ++ name ++ "."     -- no "in"!
   putStrLn answer
```

- as in `let`-syntax, `do`-blocks can also written via `do {..; ..; ..}`

Further Notes

- inside do-block, order is important; I/O actions are executed in order of appearance; result of block is result of last action
- `x <- a` is not available outside I/O actions,
  in particular there is no function of type `IO a -> a` which extracts the results of an action (of type `IO a`) without being an action itself (result type `a`)
  - once we are inside an IO action, we cannot escape
  - strict separation between purely functional code and I/O
  - when `IO a` does not appear inside type signature, we can be absolutely sure that no I/O ("side-effect") is performed
- `main :: IO ()` is the I/O action that is executed when running a compiled file via `ghc --make prog.hs` and then `./prog`

Using Pure Code Inside I/O Actions

```haskell
-- purely functional code
reply :: String -> String
reply name =
  "Pleased to meet you, " ++ name ++ ".\n" ++
  "Your name contains " ++ n ++ " characters."
  where
    n = show $ length name
-- invoked from I/O-part
main :: IO ()
main = do
  putStrLn "Greetings again. What's your name"
  name <- getLine
  let niceReply = reply name
  putStrLn niceReply
```

- invoking pure code inside I/O is easy
- the other direction is not possible

Some Predefined I/O Functions

- `return :: a -> IO a` – turn anything into an I/O action
- `System.Environment.getArgs :: IO [String]` – get command line arguments
- `putChar :: Char -> IO ()` – print character
- `putStr :: String -> IO ()` – print string
- `putStrLn :: String -> IO ()` – print string followed by newline
- `getChar :: IO Char` – read single character from stdin
- `getLine :: IO String` – read line (excluding newline)
- `interact :: (String -> String) -> IO ()` – use function that gets input as string and produces output as string
- `type FilePath = String`
- `readFile :: FilePath -> IO String` – read file content
- `writeFile :: FilePath -> String -> IO ()`
- `appendFile :: FilePath -> String -> IO ()`

Recursive I/O Actions

- branching and recursion is also possible with I/O actions
- example: implement `getLine` via `getChar`

```
get_line = do
  c <- getChar
  if c == '\n'                    -- branching
    then return ""
    else do
      line <- get_line            -- recursion
      return $ c : line
```

Examples – Imitating Some GNU Commands

- `cat.hs` – print file contents

```
import System.Environment (getArgs)
main = do
  [file] <- getArgs
  s <- readFile file
  putStr s
```

- `wc.hs` – count number of lines/words/characters in input

```
count s = nl ++ "  " ++ nw ++ "  " ++ nc ++ "\n"
  where nl = show $ length $ lines s
        nw = show $ length $ words s
        nc = show $ length s
main = interact count
```

- `uniq.hs` – omit repeated lines of input

```
import Data.List(nub)
-- nub :: Eq a => [a] -> [a]  removes repeated entries
main = interact (unlines . nub . lines)
```

- `sort.hs` – sort input lines

```
main = interact (unlines . sort . lines)
```

Laziness and I/O Actions

- consider a simple copying program

```
main = do
  [src,dest] <- getArgs
  s <- readFile src
  writeFile dest s
```

  - `readFile` and `writeFile` are lazy; hence, even large files can be copied without fully loading them into memory
- laziness might lead to problems

```
main = do
  [file] <- getArgs
  s <- readFile file
  writeFile file (map toUpper s)
```

  - since `readFile` is lazy, after `s <- readFile file` nothing is read
  - but then the same file should be opened for writing; conflict, which will result in error during execution
  - solution: more fine-grained control via file-handles which explicitly open and close files, see lecture Operating Systems

Higher-Order on I/O Actions

- ```
  foreach :: [a] -> (a -> IO b) -> IO ()
  foreach []     io = return ()
  foreach (a:as) io = do { io a; foreach as io }
  ```
- better cat.hs
  ```
  main = do
    files <- getArgs
    if null files then interact id else do
      foreach files readAndPrint
      where readAndPrint file = do
              s <- readFile file
              putStr s
  ```
  ```

The I/O-monad

- bind and do-notation are not fixed to I/O
- there exists a more general concept of monads

```
data Exp = Const Double | Div Exp Exp
eval :: Exp -> Maybe Double
eval (Const c) = return c
eval (Div exp1 exp2) = do
  x1 <- eval exp1
  x2 <- eval exp2
  if x2 == 0
    then Nothing
    else return (x1 / x2)
```

- monads won't be covered here, but they are the reason why the Haskell literature speaks about the I/O-monad

Connect Four

- aim: implement Connect Four, MB Spiele

- with textual user interface

```
0123456
.......
.XO.X..
.XOOOXO
XOXOXOX
OXXOXOO
XXOXOOX

Player X to go
Choose one of [0,1,2,3,4,5,6]
```

Connect Four: Implementation

- clear separation between
  - user interface (I/O)
    - ask for a move
    - print the current state
    - ...
  - game logic (purely functional code)
    - type to represent a state (board + next player)
    - perform a move
    - check for a winner
    - display a state as string
    - ...
  - naturally, both parts would be written as two separate modules; but to ease file-handling in exercises, everything will be in one file

Game Logic: Interface

- types: `State`, `Move` (instance of `Show` and `Read`) and `Player`
- constant `init_state :: State`
- function `show_player :: Player -> String`
- function `show_state :: State -> String`
- function `winning_player :: State -> Maybe Player`
- function `valid_moves :: State -> [Move]`
- function `drop_tile :: Move -> State -> State`

- class `Read` provides methods to convert `String`s into other types
  - `read :: Read a => String -> a`
  - `readMaybe :: Read a => String -> Maybe a`
    import of module `Text.Read` required
  - examples
    - `(read "(41,True)" :: (Integer,Bool)) = (41,True)`
    - `(read "(41,True)" :: (Integer,Integer)) = error ...`
    - `(readMaybe "1" :: Maybe Integer) = Just 1`
    - `(readMaybe "one" :: Maybe Integer) = Nothing`

User Interface

```
-- interface of game logic: init_state, show_state,
--   show_player, winning_player, valid_moves, drop_tile

main = do
  putStrLn "Welcome to Connect Four"
  game init_state

game state = do
  putStrLn $ show_state state
  case winning_player state of
    Just player ->
      putStrLn $ show_player player ++ " wins!"
    Nothing -> let moves = valid_moves state in
      if null moves then putStrLn "Game ends in draw."
      else do
        putStr $ "Choose one of " ++ show moves ++ ": "
        move_str <- getLine
        let move = read move_str
        game (drop_tile move state)
```

Game Logic: Encoding a State and Initial State

```haskell
type Tile   = Int   -- 0, 1, or 2
type Player = Int   -- 1 and 2
type Move   = Int   -- column number
data State = State Player [[Tile]]  -- list of rows

empty :: Tile
empty = 0

num_rows, num_cols :: Int
num_rows = 6
num_cols = 7

start_player :: Player
start_player = 1

init_state :: State
init_state = State start_player
  (replicate num_rows (replicate num_cols empty))
```

Game Logic: Valid Moves and Displaying a State

```haskell
valid_moves :: State -> [Move]
valid_moves (State _ rows) =
  map fst . filter ((== empty) . snd) . zip [0..]
  $ head rows

show_player :: Player -> String
show_player 1 = "X"
show_player 2 = "O"

show_tile :: Tile -> Char
show_tile t = if t == empty then '.'
  else head $ show_player t

show_state :: State -> String
show_state (State player rows) = unlines $
    map (head . show) [0 .. num_cols - 1] :
    map (map show_tile) rows
     ++ ["\nPlayer " ++ show_player player ++ " to go"]
```

Game Logic: Making a Move

```haskell
other_player :: Player -> Player
other_player = (3 -)

drop_tile :: Move -> State -> State
drop_tile col (State player rows) = State
  (other_player player)
  (reverse $ drop_aux $ reverse rows)
    where
      drop_aux (row : rows) =
        case splitAt col row of
         (first, i : last) ->
           if i == empty
             then (first ++ player : last) : rows
             else row : drop_aux rows
```

Game Logic: Winning Player

```
winning_row :: Player -> [Tile] -> Bool
winning_row player [] = False
winning_row player row = take 4 row == replicate 4 player
  || winning_row player (tail row)

transpose ([] : _) = []
transpose xs = map head xs : transpose (map tail xs)

winning_player :: State -> Maybe Player
winning_player (State player rows) =
  let prev_player = other_player player
      long_rows = rows ++ transpose rows -- ++ diags rows
    in if any (winning_row prev_player) long_rows
      then Just prev_player
      else Nothing
```

Connect Four: Final Remarks

- implementation is quite basic
  - diagonal winning-condition missing
  - crashes when invalid moves are entered
  - no iterated matches
- exercise: improve implementation

Course: Final Remarks

- interested in functional programming? then think about attending
  - term rewriting
  - program verification
- acknowledgement: I'm grateful to Christian Sternagel, in particular for giving me complete access to his course on functional programming, so that I could easily reuse several of his slides