Contents

I/O in Haskell

- We are not going to explore how Haskell supports I/O with all its side-effects, whilst also maintaining referential transparency w.r.t. the "natural" functional semantics
- First we shall look at a few file operations to note their (destructive) side-effects
- Then we shall introduce the Haskell `IO` type constructor

File I/O Operations

Open/Close

- `openFile`

    - Input: pathname, opening mode(read/write)
    - Effect: modifies filesystem by creating new file
    - Return value: handle to new file
- `hClose`
    - Input: hile handle
    - Effect: closes file indicated by handle, modifying filesystem
    - Return value: none
- Real-world items affected: filesystem, file status
- Opening Modes: `ReadMode, WriteMode, ...`

Put/Get

- `hPutChar`
    - Input: file handle, character
    - Effect: modifies file by appending the character
    - Return value: none
- `hGetChar`
    - Input: file handle
    - Effect: reads character from file current-position, which is then incremented
    - Return value: character read
- Read world items affected: contents of and positions in open files

I/O in Haskell

- Functions that do I/O (as a side-effect) use a special abstract datatype: `IO a`
- Type `IO a` denotes a "value":
    - whose evaluation produces an I/O side-effect
    - which returns a value of a type `a` when evaluated
    - such `values` are called "I/O-actions"
- I/O-actions that don't return a value hav type `IO ()`
    - Type `()` is the singleton (aka "unit") type
    - It has only one value, also written `()`
- I/O-actions are usually invoked using special syntax ("`do`-notation")

Other File I/O Types

- File opening mode

```
data IOMode = ReadMode | WriteMode | ...
```

- File Pathname - just a string

```
type FilePath = String
```

- File Handles - *pointers* to open files

```
data Handle = ...
```

- This are no types to represent file themselves, or the file-system
- I/O in Haskell works by hiding references to external data that is destructively updated

Haskell File I/O Functions

```
openFile :: FilePath -> IOMode -> IO Handle
```

- `openFile fp mode` is an I/O-action that opens a file and returns a new handle

```
hClose :: Handle -> IO ()
```

- `hClose f` is an I/O-action that closes file `f`, returning nothing

```
hPutChat :: Handle -> Char -> IO ()
```

- `hPutChar f c` is an I/O-action that writes `c` to file `f`, returning nothing

```
hGetChar :: Handle -> IO Char
```

- `hGetChar f` is an I/O-action that reads from file `f`, returning the character read

```
putChar :: Char -> IO ()
```

- `putChar c` is an I/O-action that writes `c` to `stdout`, returning nothing

```
getChar :: IO Char
```

- `getChar` is an I/O-action that reads from `stdin`, returning the character read

I/O Actions

Composing

- For this stuff to be useful we need some way of *composing* two actions together in a safe (that is, single-threaded) way
- As ever in Haskell, the solution is to come up with a function
- Perhaps something with a type like this?

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

- And we could use it like this:

```
putAB = seqIO (putChar 'a') (putChar 'b')
```

- Actually, this will read better s we make seq90 an infix function:

```
(>>) :: IO a -> IO b -> IO b
infixl 1 >>

putAB = putChar 'a' >> putChar 'b'
```

- We need something more elaborate to handle IO actions that produce results
- If we pass on the result of the first action as an input to the second then we can makeuse of it in subsequent actions

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

- Obviously we want to make this infix as well

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

- Easy to use this for simple compositions:

```
getPut = getChar >>= putChar
```

- We don't always want to make use of the result right away - one neat solution is to use a lambda abstraction:

```
swap2char
  = getChar >>=
    (\ c1 -> getChar >>=
      (\ c2 -> putChar c2 >> putChar a1))
```

Building

- Apart from the four file I/O actions just presented, we have ways to build out own

```
return :: a -> IO a
```

- `return x` is an I/O action that has no side-effect, and simply returns `x`

```
get2str :: IO String
get2str
  = getChar >>=
    (\c -> getChar >>=
      (\ d -> return [c, d]))
```

Syntactic Sugar

- We can have some syntactic sugar for the `>>` and `>>=` functions
- Consider

```
getChar >>= (\ c -> getChar >>= (\ d -> return [c, d]))
```

- We can write a multiline version, dropping brackets as

```
getChar >>= \ c ->
getChar >>= \ d ->
return [c, d]
```

Look at Haskell precedence and binding rules carefully

- We can read this as: "`getChar`", call it `c`, `getChar`, call it `d`, and return `[c, d]`"

Do-notation

- We have

```
getChar >>= \ c ->
getChar >>= \ d ->
return [c, d]
```

- Haskell has syntactic sugar for this: “do-notation”:

```
do c <- getChar
   d <- getChar
   return [c, d]
```

- Restriction: the only way to compose actions is to sequence them

Invoking Actions

- If action `act a b c` returns nothing, we simply call it `act a b c`
- If action `act a b c` returns a value we can either:
  - Simply invoke it as is (discarding the return value); `act a b c`
  - Or invoke it and bind the return value to a variable; `x <- act a b c`
- The last action in a do-expression cannot bind its return value; `actlast a b c`
  - Its return value becomes that of the entire do-expression
- Anywhere we invoke an action we can put a general Haskell expression, provided it evaluated to an I/O-action
  - E.g. `if conf then actionIfTrue else actionIfFalse`
- We can define local value using a special form of `let`-expression
  - e.g. `let v = any-expression`
    * Note there is no `in` keyword at the end
    * `any-expression` need not be an I/O-action

File I/O Examples

Read character from one file, and write to another

```
fCopyChar :: FilePath -> FilePath -> IO ()
fCopyChar fromf tof
  = do ff <- openFile fromf ReadMode
       c <- hGetChar ff
       hClose ff
       tf <- openFile tof WriteMode
       hPutChar tf c
       hClose tf
```

Notes:

- No explicit reference to filesystem
- No explicit reference to file/open file data, just reference to the file handle pointer

For comparison, here's the same function, but without the "do" notation

```
fCopyChar fromf tof
  = openFile fromf ReadMode >>= \ ff ->
    hGetChar ff             >>= \ c ->
    hClose ff               >>
    openFile tof WriteMode  >>= \ tf ->
    hPutChar tf c           >>
    hClose tf
```

- But note that despite its imperative appearance, it is just function application using `>>` and `>>=`

Read character from one file and write uppercase version to another

```
fCopyRaiseChar :: FilePath -> FilePath -> IO ()
fCopyRaiseChar fromf tof
  = do ff <- openFile fromf ReadMode
       c <- hGetChar ff
       hClose ff
       tf <- openFile tof WriteMode
       hPutChar tf (toUpper c)
       hClose tf
```

Copy *all* characters from one file, writing uppercase versions to another

```
fCopyAllChars :: FilePath -> FilePath -> IO ()
fCopyAllChars fromf tof
  = do ff <- openFile fromf ReadMode
       str <- readWholeFile ff
       hClose ff
       tf <- openFile tof WriteMode
       writeWholeFile tf (map toUpper str)
       hClose tf
```

The two utilities `readWholeFile` and `writeWholeFile`

```
writeWholeFile :: Handle -> String -> IO ()
writeWholeFile _ [] = return ()
writeWholeFile h (x:xs)
  = do hPutChar h x
    writeWholeFile h xs

readWholeFile :: Handle -> IO String
readWholeFile h
  = do eof <- hIsEOF h
       if eof then return []
         else do c <- hGetChar h
                 str <- readWholeFile h
                 return (c:str)
```

The Prelude has the following two functions

```
readFile :: FilePath -> IO String
writeFile :: FilePAth -> String -> IO ()
```

So our program can in fact be written as

```
fCopyAllChars :: FilePath -> FilePath -> IO ()
fCopyAllChar fromf tof
  = do str <- readFile fromf
       writeFile tof (map toUpper str)
```

Escape

- How do we get out of I/O actions?
- ie. is there a function with type `IO a -> a`?

```
myfun :: Int -> Int
myfun x
  = (ioescape getnum) + x
    where
      getnum :: IO Int
      getnum
        = do f <- openFile "y.dat" ReadMode
             c <- hGetChar f
             hClose f
             return (ord c)
```

Where `ioescape` was this special function

- No there isn't…
- Well actually there is: `unsafePerformIO`
- This is unsafe becasue bad use of this will break referential transparency
- i.e. a Haskell program using it is *impure*
- For use as a "backdoor" by experts