Programming Paradigms

Unit 13 — Input/Output and Error Handling

J. Gamper

Free University of Bozen-Bolzano
Faculty of Computer Science
IDSE

Outline

1. Input/Output

2. Handling Errors

Outline

1. Input/Output

2. Handling Errors

Input/Output

- Remember that Haskell is pure (referential transparency), that is:
  - Functions take inputs and compute outputs (and for the same input always the same output); nothing else happens in-between
  - In particular, they have no side-effects/external effects
    - do not modify global variables or depend on it
    - may not print anything on the screen
    - may not read from the keyboard, or filesystem, or network
- Adding IO to a purely functional language is a challenge, since IO operations are **not** functions
  - Input does not need an input parameter, and may return different values
  - Output does not return a value, but clearly has side effects: it changes the state of the output device
- If IO operations were functions, this would create problems, e.g.,
  - two calls of an input function `getchar`, `getchar` would be executed only once as the result is reused, which makes no sense for IO;
  - a fake parameter could be introduced, e.g., `getchar 1`, `getchar 1`, etc., to ensure that each call is actually executed
- As we will see, it is possible to do IO in Haskell, but it looks very different than in most other languages

IO Type and Actions/1

- The solution to I/O in Haskell is a special type, called IO
- Values of type `IO a` are called IO actions and are "descriptions of effectful computations"
- If executed, an IO action `IO a`
  - performs some effectful I/O operations (side-effect, impure), and
  - produces a return value of type a (pure)
- The description itself is safe as it has no effects: `IO` is just a description on how to produce a
  - Compare it to a cake vs. a recipe on how to make a cake

```
c ::  Cake
r ::  Recipe Cake
```

- Hence, IO actions in Haskell separate the functional ("pure") parts of a program from the non-functional ("impure") parts

IO Type and Actions/2

- Haskell has getLine to read a string and putStrLn to print a string
- Lets have a look at the types of getLine and putStrLn?

```
> :t getLine
getLine ::  IO String
```

  - getLine has no input parameter and returns an IO action
  - The IO action does some “dirty” stuff in IO, but the result is a “clean” data type, namely a string

```
> :t putStrLn
putStrLn ::  String -> IO ()
```

  - putStrLn gets a string as input parameter and returns an IO action
  - The IO action does some “dirty” stuff and returns ()

- The type () is called unit and has one value, namely () (similar to void)

Executing an IO Action

- Recall that the value of an IO action is just a recipe, which does not do anything
- But how can we actually execute IO actions?
- For an executable Haskell program, there is only one way to execute an IO action: assign it to `main`, which will run it for you

```
module Main where

main = putStrLn "Hello World!"
```

- The use of the name `main` is important: `main` is defined to be the entry point of a Haskell program (similar to the main function in C)
- Actually, `main` forwards the execution of IO actions to the Haskell runtime system
- You can put the above in a file `helloworld.hs` and run it through ghc to get an executable program

Executing Sequences of IO Actions/1

- Running a single IO action would not lead to very exciting programs
- Haskell allows you to “glue” together IO actions using the `do` notation

```
main = do
    putStrLn "Hi there, what’s your name?"
    name <- getLine
    putStrLn ("Hello " ++ name ++ "!")
```

- The lines in a `do`-block work similar to an imperative execution
  - Allows to execute a sequence of IO actions, one after the other
- `<-` extracts the “pure” part (the string) from `getLine`’s return value, which has type `IO String`
  - Can only be used in a `do`-block
- Notice that `name = getLine` and `putStrLn ("Hello " ++ getLine )` would not work

Executing Sequences of IO Actions/2

- The IO action carries along the “baggage of the impure” context
  - So you don't have to worry about it
- If you want to do a “pure” assignment in the context of IO, you have to use let

```
module Main where

import Data.Char

main = do
    putStrLn "What's your name?"
    name <- getLine
    let bigName = map toUpper name
    putStrLn ("Hi " ++ bigName ++ "!")
```

- The let statement in a do-block allows you to create a new variable bound to a “pure” value

Executing Sequences of IO Actions/3

- In summary, a do-block
  - introduces a sequence of statements
  - and executes these statements in order
- A statement can be one of the following:
  - an IO action
  - a <-, binding the ("pure") result of an action
  - a let, expressing "pure" definitions

Executing IO Actions in GHCI

- An IO action can also be executed directly in the interactive Haskell shell, like any other function

```
> putStrLn "Hello World!"
Hello World!
```

- We can also use IO functions in the body of other functions

```
> let hw = putStrLn "Hello World!"
> hw
Hello World!
```

- So, there's no need to go via `main` in the shell
- That means, in the shell we are in an `IO` environment
- Consequently, we had to use `let` to do "pure" stuff

File IO – Reading

- Lets look at file IO, using an example that counts the # of lines of a file

```
module Main where

import System.IO

main = do
    theInput <- readFile "countlines.hs"
    putStrLn (countLines theInput)

countLines ::  String -> String
countLines str = show (length (lines str))
```

- `import System.IO` is a so-called language pragma, which imports features that are not part of the standardized Haskell language
- The `readFile` function reads a file and returns the contents of the file as a string; the file is read lazily, on demand
- The function `lines :: String -> [String]` breaks a string on newline and returns an array of strings
- The function `length :: [a] -> Int` returns the length of a finite list

File IO – Writing

- Writing to a file is simple

```
module Main where

import System.IO

main = do
    putStrLn "Writing to a file ..."
    putStrLn "What do you want to write?"
    what <- getLine
    putStrLn "To which file?"
    file <- getLine
    writeFile file what
```

- `writeFile` will overwrite an existing file
- Use `appendFile` if you'd like to append instead

Monads

- The principle used for IO actions can be generalized and not only applied to IO
- Haskell uses the concept of a monad to handle "impurity"
  - For example, for IO, non-determinism, and exceptions
- We are going to introduce the general principle a bit later
- First, we are going to look at another example where Haskell meets the messy "real world"

Outline

1. Input/Output

2. Handling Errors

Handling Errors

- Sometimes things go wrong, i.e., a function is not able to return a value
- For example, if we call `head` on an empty list, we get an error
- We don't necessarily want the program to just stop working and output an error in a case like that
- However, a function always has to return a value
- So we have to be able to handle the concept of failure (which is "impure" in Haskell's eyes)

Errors and the MayBe Type

- Haskell offers the type constructor Maybe that has a type parameter a:

```
data Maybe a = Nothing | Just a
```

- Maybe a is a normal data type, but it "lifts" a data type a into a new context
- A value of type Maybe a represents a value of type a with the the context of a possible failure attached to it
  - A value of Just 1 means that the number 1 is there
  - The extra value Nothing represents the lack of value of type a or a computation failure or . . .
- The type system then requires that you check for that extra value, which prevents a remarkable number of bugs
- Many other languages handle this sort of "no-value" value with NULL

Handling Errors with the MayBe Type/1

- Now we can "wrap" the result of a function call inside of a Maybe:
  - if the function call was successful, we hand it to the value constructor Just
  - otherwise, it becomes Nothing
- Let's write an alternative version of head that can cope with empty lists

```
safeHead ::  [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x
```

```
> safeHead [1,2,3]
Just 1
```

```
> safeHead []
Nothing
```

Handling Errors with the `MayBe` Type/2

- However, this comes at a price: we've introduced "impurity" into our function
- For example, the following expression will raise an error

```
doubleMe (safeHead [1,2,3])
```

- The result of `safeHead` is `Just 1` (of type `MayBe`), but `doubleMe` expects a pure integer
- So, how can we use the "impure" result of `safeHead` in other pure functions?
- Hint: `Maybe` is an instance of the type class `Functor`
  - Quick reminder: a functor can be seen as content "wrapped" in a box
  - So, Haskell does not allow the concept of failure to escape its impure box
  - So we have to get inside of the box

Handling Errors with the `MayBe` Type/3

- The typeclass `Functor` provides the function `fmap` to get inside the "box"
- `fmap` gets us on the inside of `Maybe`

```
> fmap doubleMe (safeHead [1,2,3])
Just 2
```

- `safeHead [1,2,3]` returns `Just 1`
- `fmap` pushes the execution of `doubleMe` inside the "Just box"

```
> fmap doubleMe (safeHead [])
Nothing
```

- If there is `Nothing` inside, `fmap` will not even apply the function, but return `Nothing`