Lecture 9. Input and output

Functional Programming

Utrecht University

Big picture

- This course: typed, *purely* functional programming
- Today: purity and impurity

Goals

- Learn the difference between pure and impure
- Interact with the outside world in Haskell
  - Input/output
  - Random generation
- Introduce do- and monadic notation through an example

Chapter 10 from Hutton's book

Interactive programs

- In the old days, all programs were *batch* programs
  - Introduce the program and input, sit and drink tea/coffee for hours, and get the output
  - Programs were isolated from each other
  - The part of Haskell your have learnt up to now

Interactive programs

- In the old days, all programs were *batch* programs
  - Introduce the program and input, sit and drink tea/coffee for hours, and get the output
  - Programs were isolated from each other
  - The part of Haskell your have learnt up to now
- In this modern era, programs are *interactive*
  - Respond to user input, more like a dialogue
  - From the perspective of a program, it needs to communicate with an *outside world*
  - Examples?
  - Today: how we model this in Haskell!

Interactive programs

- In the old days, all programs were *batch* programs
    - Introduce the program and input, sit and drink tea/coffee for hours, and get the output
    - Programs were isolated from each other
    - The part of Haskell your have learnt up to now
- In this modern era, programs are *interactive*
    - Respond to user input, more like a dialogue
    - From the perspective of a program, it needs to communicate with an *outside world*
    - Examples?
    - Today: how we model this in Haskell!
- **examples?**

Purity = referential transparency

Referential transparency = you can always substitute a term by its definition without change in the meaning

Purity = referential transparency

Referential transparency = you can always substitute a term by its definition without change in the meaning

- *Inlining*:

  ```
  let x = e in ... x ... x ...
  ```

  is by definition equivalent to:

  ```
  (\x -> ... x ... x ...) e
  ```

  is by definition equivalent to:

  ```
  ... x ... x ... where x = e
  ```

  is (because we may inline) equivalent to:

  ```
  ... e ... e ...
  ```

Referential transparency

A concrete example:

```
reverse xs ++ xs
  where xs = filter p ys
```

is equivalent to:

```
reverse (filter p ys) ++ filter p ys
```

Referential transparency

A concrete example:

```
reverse xs ++ xs
  where xs = filter p ys
```

is equivalent to:

```
reverse (filter p ys) ++ filter p ys
```

Note that the second version duplicates work, but we are speaking here about the *meaning* of the expression, not its efficiency

Referential transparency: why care?

Referential transparency: why care?

- *Copying/duplication (contraction)*

  ```
  let x1 = e; x2 = e in t
  ```

  is always equivalent to:

  ```
  let x1 = e in t[x1/x2]
  ```

Referential transparency: why care?

- *Copying/duplication (contraction)*

  ```
  let x1 = e; x2 = e in t
  ```

  is always equivalent to:

  ```
  let x1 = e in t[x1/x2]
  ```

- *Discarding (weakening)*

  ```
  let x = e in t
  ```

  if `t` does not mention x, is equivalent to :

  ```
  t
  ```

Referential transparency: why care?

- *Copying/duplication (contraction)*

  `let x1 = e; x2 = e in t`

  is always equivalent to:

  `let x1 = e in t[x1/x2]`

- *Discarding (weakening)*

  `let x = e in t`

  if `t` does not mention x, is equivalent to :

  `t`

- *Commuting/reordering (exchange)*

  `let x1 = e1; x2 = e2 in t`

  is always equivalent to:

  `let x2 = e2; x1 = e1 in t`

Referential transparency

- Referential transparency decouples the meaning of the program from the order of evaluation
  - Inlining or duplicating does not change the program

Referential transparency

- Referential transparency decouples the meaning of the program from the order of evaluation
  - Inlining or duplicating does not change the program
- This has practical advantages:
  - The compiler can reorder your program for efficiency
  - Expressions are only evaluated (once) when really needed
    - This is called *lazy evaluation*
  - Paralellism becomes much easier

Side-effects

Interaction with the world in not referentially transparent!

Any examples?

Side-effects

Interaction with the world in not referentially transparent!

Any examples?

Suppose that `getChar :: Char` retrieves the next key stroke from the user

Why is

```
let k = getChar in k == k
```

not referentially transparent?

Side-effects

Interaction with the world in not referentially transparent!

Any examples?

Suppose that `getChar :: Char` retrieves the next key stroke from the user

```
let k = getChar in k == k
```

is always `True`, whereas this is not the case with

```
getChar == getChar
```

We say that `getChar` is a **side-effectful** action

- `getChar` is also called an **impure** function

Side-effects

- Many other actions have side-effects (why?)
  - Printing to the screen
  - Generate a random number
  - Communicate through a network
  - Talk to a database
- Intuitively, these actions influence the outside world
  - Key properties: we cannot dicard/duplicate/exchange the world
  - And thus we cannot substitute for free

Haskell typing of code with IO/side-effects

```
getChar  :: IO Char
getLine  :: IO String
getArgs  :: IO [String]

putChar  :: Char   -> IO ()
putStr   :: String -> IO ()
putStrLn :: String -> IO ()
.
.
.
```

Haskell typing of code with IO/side-effects

```haskell
return    :: a -> IO a
(>>=)     :: IO a -> (a -> IO b) -> IO b -- may not inline!
-- gives us 2nd, non-referentially transparent assignment!

getChar   :: IO Char
getLine   :: IO String
getArgs   :: IO [String]

putChar   :: Char   -> IO ()
putStr    :: String -> IO ()
putStrLn  :: String -> IO ()
.
.
.
```

How IO is implemented

Modelling output

Following this idea, we model an action by a function which changes the world

```
type IOCom = World -> World -- IO ()
```

Using `IOCom` we can give a type to `putChar`

```
putChar :: Char -> IOCom
putChar c world = ... -- details hidden
```

Modelling output

Following this idea, we model an action by a function which changes the world

```
type IOCom = World -> World -- IO ()
```

Using `IOCom` we can give a type to `putChar`

```
putChar :: Char -> IOCom
putChar c world = ... -- details hidden
```

How should we think of `World` *and* `putChar`*?*

Combining output actions

Executing two actions in sequence is plain composition

```
putAB :: IOCom
putAB world = putChar 'b'  (putChar 'a' world)
-- or using composition
putAB       = putChar 'b' . putChar 'a'
```

putStr, first version

`putStr s` prints the whole string to the screen

```
putStr :: String -> IOCom
putStr []     = id  -- keep the world as it is
putStr (c:cs) = putStr cs . putChar c
```

`putStrLn s` does the same, with a newline at the end

```
putStrLn s = putChar '\n' . putStr s
```

Modelling input

Our `IOCom` type is not suitable for `getChar`. Why not? Fix?

Modelling input

Our `IOCom` type is not suitable for `getChar`. Why not? Fix?

- Solution: pair the output value with the new world

```
type IO a = World -> (a, World)

getChar :: IO Char
getChar = ... -- details hidden
```

What is now the return type of `putChar`?

Modelling input

Our `IOCom` type is not suitable for `getChar`. Why not? Fix?

- Solution: pair the output value with the new world

```
type IO a = World -> (a, World)

getChar :: IO Char
getChar = ... -- details hidden
```

What is now the return type of `putChar`?

- We use the empty tuple as a dummy value

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

Combining input and output

Suppose that we want to echo a character

```
echo = putChar getChar
```

```
• Couldn't match expected type ‘Char’
              with actual type ‘IO Char’
```

Combining input and output

Let's try again with function composition

```
echo = putChar . getChar
• Couldn't match expected type ‘IO b’
              with actual type ‘Char -> IO ()’
```

```
getChar :: IO Char
        -- World -> (Char, World)
putChar :: Char -> IO ()
        -- Char -> World -> ((), World)
(.)   :: (b -> c) -> (a -> b) -> a -> c
```

Types do not fit, since b should be both `(Char, World)` – from `getChar` – and `Char` – from `putChar`

Solution: bind

`(>>=)` – pronounced "bind" – takes care of threading the world around

```
(>>=) :: IO a -> (a -> IO b) -> IO b
(f >>= g) w = ...
```

Based on the output of the first action, we choose which action to perform next

```
echo = getChar >>= \c -> putChar c
    -- also getChar >>= putChar
```

Solution: bind

`(>>=)` – pronounced “bind” – takes care of threading the world around

```
(>>=) :: IO a -> (a -> IO b) -> IO b
(f >>= g) w = g a' w' where
    (a', w') = f w
```

Based on the output of the first action, we choose which action to perform next

```
echo = getChar >>= \c -> putChar c
     -- also getChar >>= putChar
```

Uppercase input

We want to build a `getUpper` function which returns the uppercase version of the last keystroke

```
getChar :: IO Char
toUpper :: Char -> Char
```

```
getUpper = getChar >>= \c -> toUpper c
• Couldn't match expected type ‘IO Char’
              with actual type ‘Char’
```

Uppercase input

We need a way to *embed* pure computations, like `toUpper`, in the impure world

```
return :: a -> IO a
return a = ...
```

Warning! `return` is indeed a very confusing name

- Does not “break” the flow of the function
- A more apt synonym is available, `pure`

```
getUpper = getChar >>= \c -> return (toUpper c)
         -- getChar >>= return . toUpper
```

Uppercase input

We need a way to *embed* pure computations, like `toUpper`, in the impure world

```
return :: a -> IO a
return a = \w -> (a, w)
```

Warning! `return` is indeed a very confusing name

- Does not “break” the flow of the function
- A more apt synonym is available, `pure`

```
getUpper = getChar >>= \c -> return (toUpper c)
         -- getChar >>= return . toUpper
```

Preserving purity

There is no bridge back from the impure to the pure world

```
backFromHell :: IO a -> a
```

Why?

Preserving purity

There is no bridge back from the impure to the pure world

```
backFromHell :: IO a -> a
```

Why?

In this way we ensure that the outside world never "infects" pure expressions

- Referential transparency is preserved

`IO` is abstract: never see `World`

Mixing IO and recursion

When dealing with IO, we cannot directly pattern match

- We often use case expressions after (>>=)

```
mystery :: IO String
mystery = getChar >>= (\c ->
            case c of
              '\n' -> return []
              _    -> mystery >>= (\rest ->
                        return (c : rest)
                      )
            )
```

What does this code do?

Mixing IO and recursion

When dealing with IO, we cannot directly pattern match

- We often use case expressions after (>>=)

```
mystery :: IO String
mystery = getChar >>= (\c ->
            case c of
              '\n' -> return []
              _    -> mystery >>= (\rest ->
                        return (c : rest)
                      )
          )
```

What does this code do?

Working directly with (>>=) is very cumbersome!

do-notation

Luckily, Haskell has specific notation for IO

```
getLine = do c <- getChar
             case c of
               '\n' -> return []
               _    -> do rest <- getLine
                          return (c : rest)
```

Blocks for IO start with the keyword do

- <- gives a name to the result of an IO action
- The notation was chosen to “look imperative”
- <- is not referentially transparent!

Cooking `putStr`

Let us write `putStr` with the new combinators

Cooking putStr

Let us write `putStr` with the new combinators

```
putStr :: String -> IO ()
putStr []     = return ()
putStr (c:cs) = putChar c >>= (\_ -> putStr cs)
```

Cooking putStr

Let us write `putStr` with the new combinators

```
putStr :: String -> IO ()
putStr []     = return ()
putStr (c:cs) = putChar c >>= (\_ -> putStr cs)
```

What is happening is much clearer with do-notation

```
putStr :: String -> IO ()
putStr []     = return ()
putStr (c:cs) = do putChar c
                   putStr cs
```

do-notation, in general

A general do block is translated as nested `(>>=)`

```
do x1 <- a1                a1 >>= (\x1 ->
   x2 <- a2                 a2 >>= (\x2 ->
   ...          ===>         ...
   xn <- an                   an >>= (\xn ->
   expr                        expr) ... ))
```

In addition, if you don't care about a value, you can write simply `ai` instead of `_ <- ai`

Rule of thumb: do not think about `(>>=)` at all, just use do

Guess a number

```
Pick a number between 1 and 100.
Is it 50? (g = greater, l = less, c = correct)
g
Is it 75? (g = greater, l = less, c = correct)
l
Is it 62? (g = greater, l = less, c = correct)
g
Is it 68? (g = greater, l = less, c = correct)
l
Is it 65? (g = greater, l = less, c = correct)
c
Guessed
```

Guess a number

We do *binary search* over the list of numbers

- At each step, we pick the middle value as a guess

```
guess :: Int -> Int -> IO ()
guess l u
  = do let m = l + ((u - l) `div` 2)
       putStr ("Is it " ++ show m ++ "?")
       putStrLn "(g = greater, l = less, c = correct)"
       k <- getChar
       case k of
         'g' -> guess (m + 1) u
         'l' -> guess l (m - 1)
         'c' -> putStrLn "Guessed"
         _   -> do putStrLn "Press type g/l/c!"
                   guess l u
```

Guess a number, main program

When an executable written in Haskell starts, the `main` function is called

- `main` always has type `IO ()`

```
main :: IO ()
main = do (l:u:_) <- getArgs
          guess (read l) (read u)
```

- `getArgs :: IO [String]` obtains program arguments
- `read :: Read a => String -> a`
    - Parses a `String` into a value
    - In this case, we parse it into an `Int`

Summary of basic I/O actions

```
return    :: ???
(>>=)     :: ???

getChar   :: ???
getLine   :: ???
getArgs   :: ???

putChar   :: ???
putStr    :: ???
putStrLn  :: ???
```

Summary of basic I/O actions

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

getChar   :: IO Char
getLine   :: IO String
getArgs   :: IO [String]

putChar   :: Char   -> IO ()
putStr    :: String -> IO ()
putStrLn  :: String -> IO ()
```

Dealing with files

The simplest functions to work with files in Haskell

```haskell
type FilePath = String

readFile  :: ???
writeFile :: ???
```

Dealing with files

The simplest functions to work with files in Haskell

```
type FilePath = String

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

The following functions are often convenient

```
lines   :: String -> [String]  -- break at '\n'
unlines :: [String] -> String  -- join lines

-- convert back and forth
show :: Show a => a -> String
read :: Read a => String -> a
```

Guess a number, bounds from file

```
main :: IO ()
main = do -- Read the contents of the file
          config <- readFile "guess.config"
          -- Get the first two lines
          let l:u:_ = lines config
          -- Parse the numbers and start guessing
          guess (read l) (read u)
```

IO as first-class citizens

IO actions are first-class

In the same way as you do with functions

- An IO action can be an argument or result of a function
- IO actions can be put in a list or other container

```
map (\name -> putStrLn ("Hello, " ++ name))
    ["Mary", "John"]  ::  [IO ()]
```

Building versus execution of IO actions

```
map (\name -> putStrLn ("Hello, " ++ name))
    ["Mary", "John"]  ::  [IO ()]
```

Running this code prints **nothing** to the screen

- We say that it *builds* the `IO` actions: describes what needs to be done but does not do it yet

To obtain the side-effects, you need to *execute* the actions

- At the interpreter prompt
- In a do block which is ultimately called by `main`
- An executed action always has a `IO T` type

Execute a list of actions

`sequence_ as` performs the side-effects of a list of actions

1. Define the type

```
sequence_ :: [IO a] -> IO ()
```

Execute a list of actions

`sequence_ as` performs the side-effects of a list of actions

1. Define the type

```
sequence_ :: [IO a] -> IO ()
```

2. Enumerate the cases

```
sequence_ []     = _
sequence_ (a:as) = _
```

Execute a list of actions

`sequence_ as` performs the side-effects of a list of actions

1. Define the type

```haskell
sequence_ :: [IO a] -> IO ()
```

2. Enumerate the cases

```haskell
sequence_ []     = _
sequence_ (a:as) = _
```

3. Define the cases

```haskell
sequence_ []     = return ()
sequence_ (a:as) = do a
                      sequence_ as
```

Execute a list of actions

We have all the ingredients to greet a list of people

```
greet :: [String] -> IO ()
greet = sequence_
        . map (\name -> putStrLn ("Hello, " ++ name))
```

Execute a list of actions

We have all the ingredients to greet a list of people

```
greet :: [String] -> IO ()
greet = sequence_
        . map (\name -> putStrLn ("Hello, " ++ name))
```

This combination is very common, so the library defines

```
mapM_ :: (a -> IO b) -> [a] -> IO ()

greet = mapM_ (\name -> putStrLn ("Hello, " ++ name))
```

Execute a list of actions

By just flipping the order of arguments, we can write “imperative-looking” code

```
forM_ :: [a] -> (a -> IO b) -> IO ()
forM_ = flip mapM_

greet names = forM_ names $ \name ->
                putStrLn ("Hello, " ++ name)
```

Answer to a yes-no questions

`poseQuestion q` prints a question to the screen, obtains a y or n input from the user and returns it as a Boolean

```
poseQuestion :: String -> IO Bool
poseQuestion q
  = do putStr q
       putStrLn "Answer (y) or (n)"
       (k:_) <- getLine
       case k of
         'y' -> return True
         'n' -> return False
         _   -> do putStrLn "Answer (y) or (n)"
                   poseQuestion q
```

Gathering all answers

Once again, if we map over the list the actions are inside

```
map poseQuestion qs :: [IO Bool]
```

`sequence_` does not work, since it throws away the result

```
sequence :: [IO a] -> IO [a]
...
```

Gathering all answers

Once again, if we map over the list the actions are inside

```
map poseQuestion qs :: [IO Bool]
```

sequence_ does not work, since it throws away the result

```
sequence :: [IO a] -> IO [a]
sequence []     = return []
sequence (a:as) = do r  <- a
                     rs <- sequence as
                     return (r:rs)
```

Gathering all answers

Now we can gather answers to all questions at once

```
poseQuestions :: [String] -> IO [Bool]
poseQuestions = sequence . map poseQuestion
```

We have non-forgetful versions of the previous functions

```
mapM :: (a -> IO b) -> [a] -> IO [b]
forM :: [a] -> (a -> IO b) -> IO [b]
```

Naming convention: a function which ends in _ throws away information

Lifting

```
liftM2 :: (a -> b -> c)
       -> IO a -> IO b -> IO c
```

Lifting

```
liftM2 :: (a -> b -> c)
       -> IO a -> IO b -> IO c
liftM2 f ia ib = do
    a <- ia
    b <- ib
    return (f a b)
```

Randomness

Random generation

Random generation is provided by the `System.Random` module of the `random` package

```
class Random a where
  randomR :: RandomGen g => (a, a) -> g -> (a, g)
  random  :: RandomGen g =>           g -> (a, g)
```

- a is the type of value you want to generate
- g is the type of random generators
  - Usually, random generators keep some additional information called the *seed*

Generating several random numbers

If you want to generate several values, you need to keep track of the seed yourself

```
generateTwoNumbers :: RandomGen g
                   => g -> ((Int, Int), g)
generateTwoNumbers g
  = ...
```

Generating several random numbers

If you want to generate several values, you need to keep track of the seed yourself

```
generateTwoNumbers :: RandomGen g
                   => g -> ((Int, Int), g)
generateTwoNumbers g
  = let (v1, g1) = random g
        (v2, g2) = random g1  -- Use new seed
    in ((v1, v2), g2)     -- Return last seed
```

Obtaining the seed

An initial value for the generator needs external input

- We have `RandomGen` instance `StdGen`
- The following function takes care of obtaining a new seed, performing random generation and updating the seed at the end

```
getStdRandom :: (StdGen -> (a, StdGen)) -> IO a
```

- Note the use of a higher-order function to encapsulate the part of the program which needs randomness

Because of their ubiquity, the following functions are provided

```
randomRIO = getStdRandom . randomR
randomIO  = getStdRandom   random
```

Summary

- Introduced purity/referential transparency and constrasted with impurity/side-effects

Summary

- Introduced purity/referential transparency and constrasted with impurity/side-effects
- Actions with side-effects which return a value of type a are represented by `IO a`
  - Pure and impure parts are perfectly delineated
  - `a -> IO b` are "impure functions from a to b
  - The `main` in a Haskell program has type `IO ()`

Summary

- Introduced purity/referential transparency and constrasted with impurity/side-effects
- Actions with side-effects which return a value of type a are represented by `IO a`
    - Pure and impure parts are perfectly delineated
    - `a -> IO b` are ”impure functions from a to b
    - The `main` in a Haskell program has type `IO ()`
- To sequence `IO` actions, use do-notation
    - Under the hood it translates to nested `(>>=)` (bind)

Summary

- Introduced purity/referential transparency and constrasted with impurity/side-effects
- Actions with side-effects which return a value of type a are represented by `IO a`
  - Pure and impure parts are perfectly delineated
  - `a -> IO b` are ”impure functions from a to b
  - The `main` in a Haskell program has type `IO ()`
- To sequence `IO` actions, use do-notation
  - Under the hood it translates to nested `(>>=)` (bind)
- `IO` actions are first-class citizens