Introduction to Functional Programming

Input/output and generating test data

Some slides are based on Graham Hutton's public slides

GÖTEBORGS UNIVERSITET
CHALMERS

Recap previous lecture

- Defining operators
- Operator fixity and binding precedence
- Currying
- Lambda expressions
- Partial application
  - Operator sections
- A larger example: tic-tac-toe

- Announcements:
  - Questions to John: security / understandable code
  - Live code now in a GitLab repository

The heart and soul of functional programming!

Today

- Input/output (IO) in Haskell
  - Input from users
  - Reading to and writing from files
  - `do`-notation
- Generating test data using QuickCheck
  - Generator combinators
  - `do`-notation
  - Type class `Arbitrary`
  - Testing properties

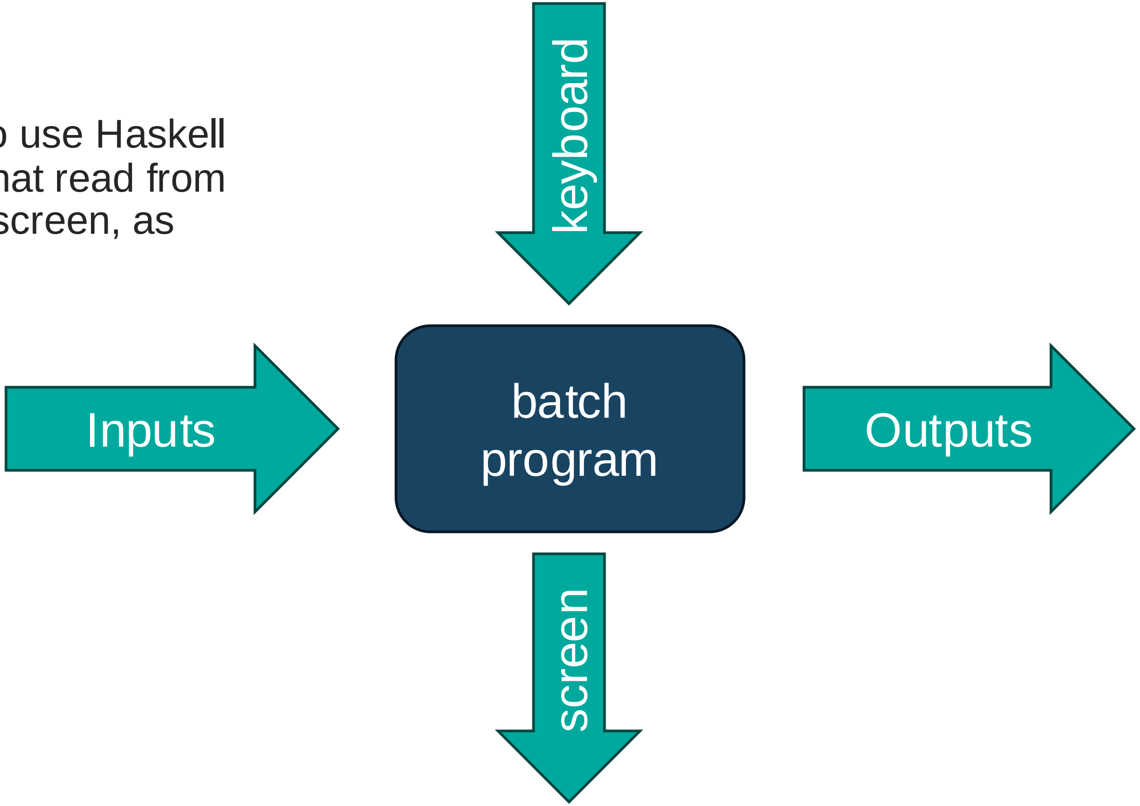
INPUT / OUTPUT

Introduction

- To date, we have seen how Haskell can be used to write batch programs that take all their inputs at the start and give all their outputs at the end.

Introduction

- However, we would also like to use Haskell to write *interactive* programs that read from the keyboard and write to the screen, as they are running.



The problem

- Haskell programs are pure mathematical functions:

Haskell programs
*have **no** side effects*!

- However, reading from the keyboard and writing to the screen are side effects:

Interactive programs
have side effects!

Pure functions

- What is a function?
  - In mathematics a function gives a single result for each input
  - In Haskell, unlike other programming languages, *functions* (like in mathematics) *always give the same result* whenever you give the same argument
- Again, there are no *side-effects*!
- In Haskell, an equation like `g x = x + 123` essentially means that we have two *equal*, *interchangeable* things
- You can use *equational reasoning* to *prove* that functions have desired properties

```
f x = g x + g (x * x)

g x = x + 123
```

```
Python
>>> f(12) - f(12) == 0
False
```

May be `True` or `False`

How would you do that?

- Suppose you wanted to model an *n*-sided die

  ```
  die :: Int -> Int
  ```

  so that `die n` gives a random number between 1 and n

```
ghci> die 6
3
ghci> die 6
4
```

- Read from a file:

  ```
  type FileName = String
  readFromFile :: FileName -> String
  ```

  given the name of a file in your computer it returns the contents of the file as a string

```
-- These functions can NOT
-- be written in Haskell

die :: Int -> Int

readFromFile :: FilePath -> String
```

Haskell instructions

- In Haskell this dilemma is solved by introducing a special type for *instructions* (sometimes called *actions* or *commands*)
- `IO Integer` (for example) is the type of *instructions* for producing an integer
- When GHCi evaluates something of type `IO t` it computes and returns the value (of type `t`) but *also* then *runs the* instructions

What is the type of `writeFile`?

```
ghci> :t writeFile
writeFile :: FilePath -> String -> IO ()
```

INSTRUCTIONS to the operating system to write the file

Just a `String`

- When you give GHCi an expression of type `IO`, it *executes the instructions*
- Note: The function `writeFile` does *not* write the file. It only computes the instruction to write

GÖTEBORGS UNIVERSITET
CHALMERS

The type `()`

- The type `()` is called the *unit type*
- It only has one value, namely `()`
- We can see `()` as the “empty tuple”
- It *means* that there is no interesting result

```
data () = ()
```

Some simple examples

- Prints the string ”alex”

```
ghci> putStrLn "alex"
alex
```

- Writes the text “Haskell FTW!” to the file called “file.txt”
  - No result displayed – why not?

```
ghci> writeFile "file.txt" "Haskell FTW!"
```

The `IO` type

- Package impure functions in the `IO` data type
- All functions that interact with the outside world in some way have `IO` in their result type
- There is no way to "hide" a call to an "impure" function inside a pure function

Look in the standard modules:
`System.IO, System.*`

```
data IO a = …   -- a built-in type

putStr :: String -> IO ()

putStrLn :: String -> IO ()

readFile :: FilePath -> IO String

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

getLine :: IO String
…
```

Using `IO` results

- The left arrow `<-` allows us to get the result of an `IO` operation
- Note the types:

```
readFile "file.txt" :: IO String
s :: String
```

  are different!

- We don't write:

```
s = readFile "file.txt"
```

  here, since it would suggest that we have two equal, interchangeable things, of the *same* type

```
ghci> s <- readFile "file.txt"

ghci> s ++ reverse s
"Hello again!!niaga olleH”
```

Combining instructions

```
catFiles :: FilePath -> FilePath -> IO String
catFiles file1 file2 = do
  s1 <- readFile file1
  s2 <- readfile file2
  return (s1 ++ s2)
```

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

- Use `do` to *combine* instructions into larger ones
- Use `return` to create an instruction with just a result
- We cannot hide a call to an `IO` function inside a pure function
- Pure functions have no side effects: same argument ⇒ same result
- Beware of the layout-rule!

Functions vs. instructions

- *Functions* always give the same result for the same arguments
- *Instructions* can behave differently on different occasions
- Confusing them is a major source of bugs!
- Most programming languages do so...
  - ...understanding the difference is important!
- But Haskell still has impure functions for `IO`, but impure functions are *kept apart*, by using the `IO` type to mark impure functions
- You can’t hide the fact you are doing `IO`
  - Once you enter the impure world, there is no way back to the pure world

Why purity?

- *Purity is good for modularity and simplifies debugging and testing*
  - Unexpected side effects and hidden dependencies is a major source of bugs
  - Automated testing with QuickCheck is an example where it is a good that functions are pure!
  - Subexpressions can be computed in any order, even in *parallel*, without changing the result of the program
- *Good functional programming style:*
  - Organize the code so that it consists mostly of pure functions.
  - Minimize the amount of code that uses `IO`

GENERATING TEST DATA

Random testing with QuickCheck

- QuickCheck generates 100 random integers. How is this done?

```
prop_example :: Integer -> Bool
prop_example n = (n+3)^2 == n^2 + 6*n + 9
```

```
> quickCheck prop_example
+++ OK, passed 100 tests.
```

Test data generators

- QuickCheck uses a type `Gen a` for random test data generators
- The `Abitrary` type class provides an *overloaded* function with instances available for most predefined types
- You can have a look at the random values generators produce

```
data Gen a = …
```

```
arbitrary :: Arbitrary a => Gen a
```

```
sample    :: Show a => Gen a -> IO ()
sample'   :: Gen a -> IO [a]
generate :: Gen a -> IO a
```

Sampling test data

- Same argument, but different results
- Starting ’small’ and increasing in size

```
ghci> sample' (arbitrary :: Gen Integer)
[0,2,-1,-6,2,4,4,-11,14,9,3]

ghci> sample' (arbitrary :: Gen Integer)
[0,2,-2,-3,5,4,-9,-5,16,-13,-6]

ghci> sample (arbitrary :: Gen [Integer])
[]
[2]
[-3]
[4]
[7,-2,-3,-8,5,8,-5]
[3,-4]
[-10,0,7,1,2,5,-12,-3,8,-7,10,11]
[-2,3,-5,-8,-3,-2,14,6,1,6,7,-2,-3,-12]
```

Functions for creating generators

- From the QuickCheck library:

```
elements :: [a] -> Gen a

oneof :: [Gen a] -> Gen a

frequency :: [(Int, Gen a)] -> Gen a

listOf :: Gen a -> Gen [a]

vectorOf :: Int -> Gen a -> Gen [a]

choose :: Random a => (a, a) -> Gen a
```

Sampling generators

- Testing `choose` and `listOf`

```
> sample' (choose ('a', 'z'))
"ozlikagbygw"

> sample' (listOf (choose ('a', 'z')))
["","t","xviy","","hfpkft","","iiisswjvrkg","suoz","slfosefhofpgla","","l"]
```

Generating a Suit

- Remember the data type `Suit`?

```
data Suit = Spades | Hearts | Diamonds | Clubs
  deriving (Eq, Show)
```

- The function `elements` is perfect for generating random suits:

```
genSuit :: Gen Suit
genSuit = elements [Spades, Hearts, Diamonds, Clubs]
```

Generating a `Rank`

- We could create a generator with `elements` and a list of all ranks
- The `frequency` *combinator* gives us more flexibility
- `genRank` and `genRank’` have different distributions

```
data Rank = Numeric Integer
          | Jack | Queen | King | Ace
          deriving (Eq, Show)
```

```
genNumeric :: Gen Rank
genNumeric =
  elements [Numeric n | n <- [2..10]]

genRoyal :: Gen Rank
genRoyal =
  elements [Jack, Queen, King, Ace]

genRank :: Gen Rank
genRank = oneof [genNumeric, genRoyal]

genRank’ :: Gen Rank
genRank’ = frequency [ (9, genNumeric)
                     , (4, genRoyal) ]
```

Generating a `Card`

- Here we want to reuse `genSuit` and `genRank`
- How can we do what? Are there some functions to combine generators in the QuickCheck library?

```
data Card = Card Rank Suit
  deriving (Eq, Show)


genCard :: Gen Card
genCard = …
```

`Gen` vs `IO`

- The type `Gen a` is for functions that generate random values of type `a`
- The type `IO a` is for functions that performs `IO` instructions and return values of type `a`
- But both are *monads,* and the `do`-notation can be used with any monad!
  - We can build larger generator functions from smaller ones in the same way as we build larger `IO` functions from smaller ones.

Generating a `Card`

```
data Card = Card Rank Suit
  deriving (Eq, Show)

genCard :: Gen Card
genCard = do
  r <- genRank
  s <- genSuit
  return (Card r s)
```

More examples

```
genEven :: Gen Integer
genEven = do
  n <- arbitrary
  return (2*n)


genNonNegative :: Gen Integer
genNonNegative = do
  n <- arbitrary
  return (abs n)
```

Telling QuickCheck which generator to use

- The `Rank` type contains values that are not valid ranks

- Two ways to use generators:
  - Using the function `forAll`
  - Using `arbitrary`

```
validRank :: Rank -> Bool
validRank (Numeric n)  = 2 <= n && n <= 10
validRank _            = True


prop_all_validRank_1 =
  forAll genRank validRank


instance Arbitrary Rank where
  arbitrary = genRank


prop_all_validRank_2 r = validRank r
```

Arbitrary

- It specifies the *default* test data generator for each type
- It makes it convenient to test a property with `quickCheck`
  - We don't need to say which test data generator to use
- The function `forAll` lets us use other generators

```
class Arbitrary where
  arbitrary :: Gen a
  shrink :: a -> [a]


instance Arbitrary Bool
instance Arbitrary Int
instance Arbitrary Integer

-- many more instances...
```

Test data distribution

- How do we know if a test has tested all the important cases?
- Using `collect r` doesn't change the test, but *collects* the values of `r` that were tested

```
prop_all_validRank_3 r =
  collect r (validRank r)
```

```
> quickCheck prop_all_validRank_3
+++ OK, passed 100 tests:
12% Numeric 10
11% Numeric 7
10% Ace
9% Numeric 6
9% Numeric 4
9% King ...
1% Jack
```

Testing `insert`

- The function `insert` inserts a value at the right place in an *ordered* list
- The output list is ordered if the input list is ordered
- How do we test it?

```
> insert 'c' "abdef"
"abcdef"
```

Testing `insert`, first try

- If the input list isn't ordered, the output list won't be ordered: bad test

```
prop_insert_1 :: Int -> [Int] -> Bool
prop_insert_1 x xs =
  isOrdered (insert x xs)
```

```
> quickCheck prop_insert_1
*** Failed! Falsifiable (after 3 tests
and 4 shrinks):
0
[0,-1]
```

Testing `insert`, second try

- The probability that a random list is ordered is *very low*
  - (can we check this?)

```
prop_insert_2 :: Int -> [Int] -> Property
prop_insert_2 x xs =
  isOrdered xs ==> isOrdered (insert x xs)
```

```
> quickCheck prop_insert_2
*** Gave up! Passed only 68 tests.
```

- We need a test generator for ordered lists!

Testing `insert`, third try

- Introduce a data type for sorted lists
- Make an instance for this data type
- Use it in a property

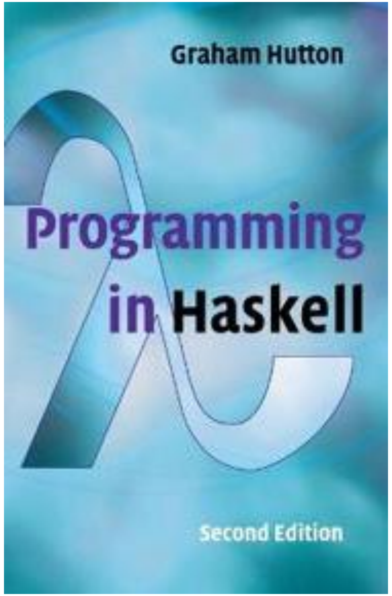
```
data SortList a = SL [a] deriving (Eq, Show)


instance (Arbitrary a, Ord a) =>
         Arbitrary (SortList a) where
  arbitrary = do
    xs <- arbitrary
    return (SL (sort xs))


prop_insert'' :: Int -> SortList Int -> Bool
prop_insert'' x (SL xs) =
  isOrdered (insert x xs)
```

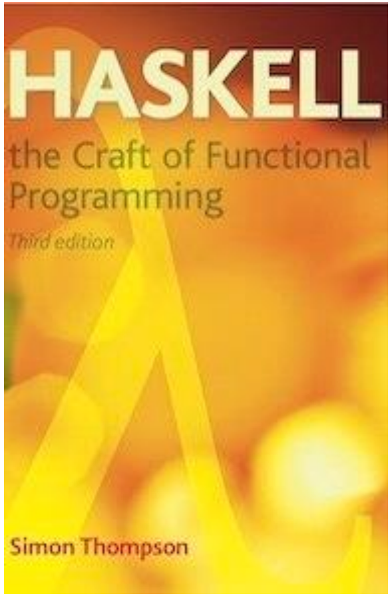
Testing `insert`, fourth try

- Fortunately, QuickCheck has a (type level) modifier `OrderedList`
- You can add a type signature to indicate you want to use an ordered list.

```
prop_insert_4 :: Int -> OrderedList Int -> Bool
prop_insert_4 x xs = isOrdered (insert x (getOrdered xs))
```

- You must use the function `getOrdered` to extract the actual list





Reading suggestions

- Hutton:
  - Chapter 10.1 – 10.6, the remainder of the chapter is optional
- Thompson:
  - Chapter 8
- Both books offer many exercises!

GÖTEBORGS
UNIVERSITET

CHALMERS