Lecture 1. FP? Haskell?

Functional Programming 2019/20

Alejandro Serrano

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Goals

- What is Functional Programming?
- Why Functional Programming? Why Haskell?
- How do I “run” Haskell?

Chapters 1 and 2 from Hutton’s book

What is Functional Programming?

More a *style* than a *paradigm*

- You can write “functional code” in almost any language

What is Functional Programming?

More a *style* than a *paradigm*

- You can write "functional code" in almost any language

Some distinguishing **features**:

1. Recursion instead of iteration
2. Pattern matching on values
3. Expressions instead of statements
4. Functions as first-class citizens

C# vs. Haskell

```
int sumUpTo(int n) {
  int total = 0;
  for (int i = n; n > 0; i--)
    total += i;
  return total;
}
```

```
sumUpTo 0 = 0
sumUpTo n = n + sumUpTo (n-1)
```

Try it!

1. Open your laptop (or look at your peer's)
2. Go to `http://repl.it/languages/haskell`
3. Write your definitions on the left pane

```
sumUpTo 0 = 0
sumUpTo n = n + sumUpTo (n-1)
```

4. Click *Run*
5. Execute your functions on the right pane

```
> sumUpTo 3
6
```

Recursion instead of iteration

Iteration = repeating a process a number of times

```
int sumUpTo(int n) {
  int total = 0;
  for (int i = n; n > 0; i--)
    total += i;
  return total;
}
```

Recursion = defining something in terms of itself

```
sumUpTo 0 = 0
sumUpTo n = n + sumUpTo (n-1)
```

Pattern matching on values

A function is defined by a series of **equations**

- The value is compared with each left side until one “fits”
- In `sumUpTo`, if the value is zero we return zero, otherwise we fall to the second one

```
sumUpTo 0 = 0
sumUpTo n = n + sumUpTo (n-1)
```

Expressions instead of statements

What code **does** versus what code **is**

- Statements manipulate the **state** of the program
- Statements have an inherent **order**
- *Variables* name and store pieces of state

```
int sumUpTo(int n) {
  int total = 0;
  for (int i = n; n > 0; i--)
    total += i;
  return total;
}
```

Expressions instead of statements

What code **does** versus what code **is**

- Value of a *whole expr.* depends only on its *subexpr.*
- Easier to compose and **reason** about
  - We will learn how to reason about programs

```
sumUpTo 3   -->   3 + sumUpTo 2
            -->   3 + 2 + sumUpTo 1
            -->   ...
```

Functions as first-class citizens

Function = mapping of arguments to a result

```haskell
-- In the left pane
greet name = "Hello, " ++ name ++ "!"
```

- Functions can be parameters of another function
- Functions can be returned from functions

```haskell
-- In the right pane
> map greet ["Mary", "Joe"]
["Hello, Mary!", "Hello, Joe!"]
```

`map` applies the function `greet` to each element of the list

Try it yourself!

Build `greet` with two arguments

```
> greet "morning" "Paul"
"Good morning, Paul!"
```

```
-- Here is the version with one argument
greet name = "Hello, " ++ name ++ "!"

_
```

Why Functional Programming?

To create **better** software

To create better software

1. Short term: fewer bugs
   - *Types* prevent the “stupid” sort
     - What does `True + "1"` mean?
   - *Purity* means fewer surprises when programming
     - A function can no longer mutate a global state
   - *Purity* makes it easier to *reason* about programs
     - Reasoning about OO $\Longrightarrow$ master/PhD course
     - Reasoning about FP $\Longrightarrow$ this course

To create better software

2. Long term: more maintainable
   - Higher-order functions *remove* lots of *boilerplate*
     - Also, less code to test and fewer edge cases
   - Types are *always updated* documentation
   - Types help a lot in *refactoring*
     - Change a definition, fix everywhere the compiler tells you there is a problem

FP is gaining traction

- F# for .NET, Scala and Kotlin for JVM, Swift for iOS
- C# and Java are getting functional constructs

```
string greet(string name) {
  return "Hello, " + name + "!"
}

var names = new List() { "Mary", "Joe" }
names.Map(x => greet(x))
```

- Haskell has a growing user base and nice ecosystem
  - You can do webdev in Haskell!

Why Haskell?

Haskell can be defined with four adjectives

- Functional
- Statically typed
- Pure
- Lazy

Haskell is statically typed

- Every expression and function has a *type*
- The compiler *prevents* wrong combinations

```
> :t (+)  -- Give me the type of +
Int -> Int -> Int
> 1 + 2
3
> 1 + True
Couldn't match expected type ‘Int’
                with actual type ‘Bool’
```

Inference = if no type is given for an expression, the compiler *guesses* one

Haskell is pure

- You **cannot** use statement-based programming
  - Variables do not change, only give names
  - Program is easy to compose, understand and paralellize
- Functions which interact with the “outer world” are marked in their type with `IO`
  - This prevents unintended side-effects

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

Haskell is lazy

We shall get to this one…

Why Haskell?

From a pedagogical standpoint

- Haskell *forces* a functional style
  - In contrast with imperative and OO languages
  - We can do *equational reasoning*
- Haskell teaches the value of static types
  - Compiler finds bugs long before run time
  - We can express really detailed invariants

How do I “run” Haskell?

GHC

- We are going to use GHC in this course
  - The (Glorious) **G**lasgow **H**askell **C**ompiler
  - State-of-the-art and open source
- Windows and Mac
  - Go to `https://www.haskell.org/downloads`
  - Install *Haskell Platform Full*
- Linux
  - `sudo pkg-mgr install haskell-platform`
  - where `pkg-mgr` is your package manager: `apt-get`, `yum`, `emerge`, ...

Compiler versus interpreter

- Compiler (`ghc`)
  - Takes one or more files as an input
  - Generates a library or complete executable
  - There is **no interaction**
  - How you do things in *Imperatief/Mobiel/Gameprogrammeren*
- Interpreter (`ghci`)
  - **Interactive**, expressions are evaluated on-the-go
  - Useful for **testing** and **exploration**
  - You can also *load* a file
    - Almost as if you have typed in the entire file
  - `repl.it` is web-based `ghci`

GHC interpreter, `ghci`

1. Open a command line, terminal or console
   - Right now, just `repl.it`
2. Write `ghci` and press ⏎

   ```
   GHCi, version 8.0.1: http://www.haskell.org/ghc/
   Prelude>
   ```

3. Type an expression and press ⏎ to evaluate

   ```
   Prelude> 2 + 3
   5
   Prelude>
   ```

4. Ctrl + D (⌘ + D in Mac) or `:q` ⏎ to quit

   ```
   Prelude> :q
   Leaving GHCi.
   ```

First examples

```
> length [1, 2, 3]
3
> sum [1 .. 10]
55
> reverse [1 .. 10]
[10,9,8,7,6,5,4,3,2,1]
> replicate 10 3
[3,3,3,3,3,3,3,3,3,3]
> sum (replicate 10 3)
30
```

- Integer numbers appear as themselves
- `[1 .. 10]` creates a list from 1 to 10
- Functions are called (applied) **without** parentheses
  - In contrast to `replicate(10, 3)` in other languages

More about parentheses

- Parentheses delimit subexpressions
  - `sum (replicate 10 3)`: `sum` takes 1 parameter
  - `sum replicate 10 3`: `sum` takes 3 parameters

```
> sum replicate 10 3
<interactive>: error:
    • Couldn't match type '[t0]' with 't1 -> t'
      Expected type: Int -> t0 -> t1 -> t
        Actual type: Int -> t0 -> [t0]
```

```
> sum (replicate 10 3)
30
```

First examples of types

```
> :t reverse
reverse :: [a] -> [a]
> :t replicate
replicate :: Int -> a -> [a]
```

- `->` separates each argument and the result
- `Int` is the type of (machine) integers
- `[Something]` declares a list of `Somethings`
  - For example, `[Int]` is a list of integers
- `[a]` means list of *anything*
  - Note that `a` starts with a lowercase letter
  - `a` is called a *type variable*

Operators

```
> [1, 2] ++ [3, 4]
[1, 2, 3, 4]
> (++) [1, 2] [3, 4]
> :t (++)
(++) :: [a] -> [a] -> [a]
```

- Some names are completely made out of symbols
  - Think of +, ∗, &&, ||, ...
  - They are called **operators**
- Operators are used *between* the arguments
  - Anywhere else, you use parentheses

Question

What happens if we do?

```
> [1, 2] ++ [True, False]
```

Question

What happens if we do?

```
> [1, 2] ++ [True, False]
```

Type error!

Define a function in the interpreter

```
> average ns = sum ns `div` length ns
> average [1,2,3]
2
> :t average
average :: Foldable t => t Int -> Int
```

- Functions are defined by one or more **equations**
- You turn a function into an operator with backticks
- Naming requirements
  - Function names must start with a lowercase
  - Arguments names too
- GHC has *inferred* a type for your function

Define a function in a file

You can write this definition in a file

```
average :: [Int] -> Int
average ns = sum ns `div` length ns
```

and then load it in the interpreter

```
> :load average.hs
[1 of 1] Compiling Main ( average.hs, interpreted )
> average [1,2,3]
2
```

or even work on it an then reload

```
> :r
[1 of 1] Compiling Main ( average.hs, interpreted )
```

Define a function by cases

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)
```

- Each equation goes into its own line
- Equations are checked in order
  - If `n` is 0, then the function equals 1
  - If `n` is different from 0, then it goes to the second
- *Good style*: always write the type of your functions

Question

What happens if we write?

```
fac :: Int -> Int
fac n = n * fac (n-1)
fac 0 = 1
```

More basic types

- `Bool`: `True` or `False` (note the uppercase!)
  - Usual operations like `&&` (and), `||` (or) and `not`
  - Result of comparisons with `==`, `/=`, `<`, ...
  - **Warning!** `=` defines, `==` compares

```
> 1 == 2 || 3 == 4
False
> 1 < 2 && 3 < 4
True
> nand x y = not (x && y)
> nand True False
True
```

More basic types

- `Char`: one single symbol
  - Written in *single* quotes: `'a'`, `'b'`, ...
- `String`: a sequence of characters
  - Written in *double* quotes: `"hello"`
  - They are simply `[Char]`
    - All list functions work for `String`

```
> ['a', 'b', 'c'] ++ ['d', 'e', 'f']
"abcdef"
> replicate 5 'a'
"aaaaa"
```

First example of higher-order function

```
> map fact [1 .. 5]
[1,2,6,24,120]
> map not [True, False, False]
[False,True,True]

> :t map
map :: (a -> b) -> [a] -> [b]
```

- `map` takes *two* arguments
  - The first argument is a function `a -> b`
  - The second argument is a list `[a]`
- `map` works for every pair of types `a` and `b` you choose
  - We say that `map` is *polymorphic*

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Homework

1. Install GHC in your machine
2. Try out the examples
3. Define some simple functions
   - Sum from `m` to `n`
   - Build `greeter` with two arguments

     ```
     > greeter "morning" ["P", "Z"]
     ["Good morning, P!", "Good morning, Z!"]
     ```
4. Think about the types of those functions

Two pieces of advice

Get yourself a good editor

- At the very least, with syntax highlighting
- Visual Studio Code and Atom are quite nice
    - Available at `code.visualstudio.com` and `atom.io`
    - Install Haskell syntax highlighting afterwards
- vi or Emacs for the adventurous

Get comfortable with the command line

- `https://tutorial.djangogirls.org/en/intro_to_command_line/`

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

A bit of history

Functional programming is quite **old**

- 1930s: Alonzo Church develops λ-calculus
  - Theoretical basis for functional languages
- 1950s: McCarthy develops Lisp, the first FP language
  - Lisp supports both statements and expressions
  - Still in use: Common Lisp, Scheme, Racket
- 1970s: ML introduces type systems with inference
- 1990s: development of the Haskell language
  - 1998: first stable version, Haskell'98
- 2010: current version of the Haskell language