Introduction to the course and basics of functional programming

Lecture 1 of CSE 3100
Functional Programming

Jesper Cockx

Q3 2023-2024

Technical University Delft

Today's lecture

- Welcome to the course
- What is functional programming?
- Installing GHC
- Haskell basics

About the course

First, a bit about me

- I'm assistant professor in the Programming Languages group.
- I do research into dependent type systems and am part of the team working on the Agda programming language.
- I'm originally from Belgium, and have been in Delft for 4 years.
- I speak Dutch & English.

Teaching team

- Jesper Cockx (lecturer)
- Ivar de Bruin (lab responsible)
- Bohdan Liesnikov (project responsible)
- Casper Dekeling (teaching assistant)
- Matej Havelka (teaching assistant)
- Anna Kalandadze (teaching assistant)
- Davis Kažemaks (teaching assistant)
- Duuk Niemandsverdriet (teaching assistant)
- Miloš Ristić (teaching assistant)
- Pallabi Sree Sarker (teaching assistant)
- Giulio Segalini (teaching assistant)

Course materials and resources

- Book: Programming in Haskell by Graham Hutton (second edition)
- Brightspace: news + slides + additional lecture notes
- Weblab: weekly exercises + final project
- Queue: asking for help during labs
- Answers: asking questions (any time)
- Course email: `fp-cs-ewi@tudelft.nl`

Lectures (Monday + Tuesday)

Week 1-4: Functional programming in Haskell

- Week 1: Haskell basics
- Week 2: polymorphism and data types
- Week 3: type classes and functors
- Week 4: monads and lazy evaluation

Week 6-7: Dependently typed programming in Agda

- Week 6: Agda and dependent types
- Week 7: theorem proving in Agda

Week 8-9: Summary + working on the project

Lab sessions (Monday + Wednesday)

Each week there will be *ungraded* programming exercises on Weblab

- Tests are run automatically
- You can submit as often as you want
- Ask questions via Queue or on Answers

See Brightspace for links and more info

Assessment

Individual project (50% of final grade)

- Assignment in three parts (week 3, 4, and 5)
- Deadline for submission: **19th of April**
- Repair option in Q4, but grade is capped at **6.0/10.0**.

Final exam (50% of final grade)

- On-campus digital Weblab exam
- Allowed material: course book, Haskell documentation
- Resit in mid-Q4

Giving feedback

This is the fourth run of this course, things have improved but are still not perfect.

If you have any feedback on the organization and/or contents of the course, please send a mail to `fp-cs-ewi@tudelft.nl`.

I will do my very best to improve things based on your feedback!

Response to feedback of last year

Feedback on the lectures, Weblab assignments, and the project was generally positive.

Main points for improvement:

- High workload for the project
  $\Rightarrow$ Updated description of requirements + test cases
- Not enough TAs in final weeks
  $\Rightarrow$ Ivar joined as extra support for coordinating TAs
- Poor support for Haskell/Agda on Weblab
  $\Rightarrow$ We are rolling out (experimental) LSP support for Haskell
- Many technical problems during exam
  $\Rightarrow$ We are working on a solution

Introduction to Functional Programming

- What is functional programming?
- Why would you study functional programming?
- Have you done functional programming before? In what languages?
- What is a functional programming language?
- What is a *pure* functional programming language?

What is functional programming?

In imperative programming, computation happens primarily by updating values:

```
sum = 0;
for (int i = 1; i <= 10; i++) {
  sum = sum + i;
}
```

In functional programming, computation happens primarily by applying functions:

```
sum [1..10]
```

Why study functional programming?

- Functional code is (often) shorter
- Functional code is (often) easier to understand
- Functional code is (often) easier to refactor
- It is (often) easier to reason about correctness of functional code
- Writing functional programs will teach you new ways to think about programming
- Functional programming languages are at the cutting edge of PL design

Also: functional programming is (or can be) fun!

What is a functional language?

Functional programming provides a toolbox of techniques for writing functional programs:

- Lambda expressions
- Higher-order functions
- Algebraic datatypes
- Pattern matching
- Recursion
- Immutable data
- Pure functions
- Lazy evaluation
- Type classes
- Monads

A ‘functional programming language’ is any language that allows us to use these tools!

What is a pure functional language?

What can happen when we call a function?

- It can return a value
- It can modify a (global) variable
- It can do some IO (read a file, write some output, …)
- It can throw an exception
- It can go into an infinite loop
- …

In a **pure** functional language (like Haskell), a function can only return a value or loop forever.

What is Haskell?

Haskell is a statically typed, lazy, purely functional programming language.

- **Static typing** means all types are checked at compile time.[1]
- **Lazy evaluation** means inputs are evaluated as needed.
- **Purity** means functions do not have side effects.

[1] Static typing $\neq$ explicit type annotations: Haskell can infer types automatically!

What is Agda?

Agda is a dependently typed, total functional programming language and a proof assistant:

- **Dependent types** are types that depend on program expressions.
- **Totality** means functions are defined and terminating for all inputs.
- **Proof assistants** allow you to state and prove properties of your programs.

The Glasgow Haskell Compiler

The Glasgow Haskell Compiler

GHC (the Glasgow Haskell Compiler) is the most popular and modern Haskell compiler.

GHCi is the interactive mode of GHC where you can type-check and evaluate Haskell expressions.

GHCup is a tool for installing GHC and related tools.

Installation: See instructions on Brightspace.

GHCi: interactive mode of GHC

Some useful commands for GHCi:

```
> 1 + 1    -- evaluate an expression
2
> :t True  -- get type of expression
True :: Bool
> :l MyFunctions.hs  -- load file
[1 of 1] Compiling Main
Ok, one module loaded.
> :q        -- quit GHCi
Leaving GHCi.
```

Writing Haskell files

A Haskell file has extension `.hs` and consists of definitions of functions and types:

```
-- file MyFunctions.hs
double x = x + x
quadruple x = double (double x)

password = "123456"
checkPassword x = x == password
```

Hello world in Haskell

A Haskell program is a script with a `main` function:

```
-- file HelloWorld.hs
main = putStrLn "Hello, world!"
```

We can compile `HelloWorld.hs` using GHC:

```
$ ghc HelloWorld.hs
[1 of 1] Compiling Main
Linking HelloWorld ...
$ ./HelloWorld
Hello, world!
```

The GHC ecosystem

Apart from the GHC compiler itself, there are many other tools for writing Haskell code:

- Hackage is a repository of Haskell libraries.
- Hoogle is a powerful search engine for Hackage.
- Cabal and Stack[2] are two build tools for building Haskell projects.
- The Haskell Language Server (HLS) provides editor services.

[2] You will use Stack for the course project.

A sneak peak at the power of Haskell

Some programs to demo (section 1.5 in the book):

- `sum`: Summing a list of numbers
- `qsort`: Sorting values
- `seqn`: Sequencing a list of actions

Haskell syntax

Function application in Haskell

Instead of `f(x)`, Haskell uses the syntax `f x`.

Mathematics	Haskell
`f(x)`	`f x`
`f(x,y)`	`f x y`
`f(g(x))`	`f (g x)`
`f(x,g(y))`	`f x (g y)`
`f(x)g(y)`	`f x * g y`

Naming rules

- Names of constructors start with a capital letter (`True`, `False`, ...)
- Names of functions and variables start with a small letter (`not`, `length`, ...)
- Names of concrete types start with a capital letter (`Bool`, `Int`, ...)
- Names of type variables start with a small letter (`a`, `t`, ...)

Reserved keywords (`if`, `let`, `data`, `type`, `module`, ...) are not allowed as names.

Haskell comments

```haskell
-- This is a single-line comment
x = 42    -- It can appear in-line
-- Each line must start with --

{- This is a multi-line comment
It starts with {- and ends with -}
Comments can be {- nested -}
-}
```

Conditional expressions

Conditionals can be expressed with `if ... then ... else ...`

```
abs x = if x < 0 then -x else x


signum x =
  if x < 0 then -1 else
    if x == 0 then 0 else 1
```

- The `else` branch is required
- `if`/`then`/`else` can be nested

Let bindings

We can give a name to a subexpression using a let binding:

```
cylinderSurfaceArea r h =
  let sideArea = 2 * pi * r * h
      topArea  = pi * r ^2
  in  sideArea + 2 * topArea
```

Anatomy of a Haskell function

```
doubleSmallNumber :: Int -> Int
doubleSmallNumber x =
    if isSmall x then 2*x else x
  where
    isSmall x = x < 10
```

- Each definition can have an (optional) type signature `f :: ...`
- A definition consists of one or more clauses `f x = ...`
- Helper functions can be put in a `where` block

`where` vs `let`

Two differences between `let` and `where`:

- `let` defines variables before they are used, `where` defines variables after they are used.
- `let ... in ...` can appear anywhere in an expression, while `where` is always attached to a clause.

Which one you use depends on the situation and your personal preference.

The Layout Rule (1/3)

Unlike C or Java, Haskell is layout-sensitive: indentation of code matters!

Definitions at the same level of indentation belong to the same block:

```
a = b + c
  where
    b = 1
    c = 2
d = a * 2
```

The Layout Rule (2/3)

Layout rule: Within a block, all definitions must start at the exact same column.

Blocks start with one of the keywords `where`, `let`, `case`, or `do`.

The Layout Rule (3/3)

Bad:

```
a = b + c
  where
    b = 1
      c = 2
```

Also bad:

```
a = b + c
  where
    b = 1
  c = 2
```

Good:

```
a = b + c
  where
    b = 1
    c = 2
```

OK but discouraged:

```
a = b + c
  where b = 1
        c = 2
```

Types in Haskell

What is a type?

A type is a name for a collection of values.

Example: `Bool` has two values `True` and `False`.

Haskell syntax for typing: `True :: Bool`

You can write type annotations (almost) anywhere in your Haskell program.

What is type inference?

In Haskell type annotations are optional:
the compiler will try to infer the type if it is not given.

In GHCi, we can ask the type of an expression with "`:t`"

```
> :t True
Bool
```

Interpreting Haskell type errors (1/2)

```
> not 't'
<interactive>:1:5: error:
• Couldn't match expected type ‘Bool’ with
  actual type ‘Char’
• In the first argument of ‘not’, namely ‘'t'
  In the expression: not 't'
  In an equation for ‘it’: it = not 't'
```

Haskell tells us that `not` takes an argument of type **`Bool`**, but was given an argument of type **`Char`** instead.

Interpreting Haskell type errors (2/2)

```
> 5 + True
<interactive>:2:1: error:
• No instance for (Num Bool) arising from
  a use of ‘+’
• In the expression: 5 + True
In an equation for ‘it’: it = 5 + True
```

Haskell tells us that `+` works on any type that implements the `Num` typeclass, but `Bool` (the type of `True`) does not.

When you get a type error, don't ignore it but try to understand what GHC is telling you!

Basic Haskell types

- `Bool`: `True` and `False`
- `Int`: `0`, `42`, `-9000`, ...(64 bit integer)
- `Integer`: `92233720368547758080`, ...(arbitrary-size integers)
- `Float`: `1.23`, `Infinity`, `NaN`, ...
- `Double`: `1.0e40`, `-1.0e300` ...
- `Char`: `'a'`, `'Z'`, `'/'`, ...
- `String`: `"Hello, world!"`, `"line 1\nline 2"`

Function types `a -> b`

Examples:

```
not :: Bool -> Bool
isDigit :: Char -> Bool
```

In general, `a -> b` is the type of (pure) functions from `a` to `b`.

A function of type `a -> b` can be applied to an argument of type `a` to get a result of type `b`:

```
> isDigit '5'
True
```

Tuple types

`(a,b)` is the type of tuples `(x,y)` where `x :: a` and `y :: b`.

Examples:

```
(42,True) :: (Int,Bool)
('x','y','z') :: (Char,Char,Char)
```

Warning: `(a,b,c)` is *not* the same as `((a,b),c)` or `(a,(b,c))`

Functions with several arguments

Two ways to define a function with 2 arguments:

```
add1 :: (Int,Int) -> Int
add2 :: Int -> (Int -> Int)
```

`add1` takes as argument a tuple `(x,y)` and returns an `Int`.

`add2` takes as argument an `Int` and returns a function of type `Int -> Int` that can be applied to the second argument.

Working with curried functions

A function that returns another function is called a curried function, after Haskell B. Curry.

```
> :t add2 1
Int -> Int
> (add2 1) 1
2
```

As a convention, Haskell functions with multiple arguments are usually curried.

Syntax conventions

The function arrow `->` associates to the right:

```
   Int -> Bool -> Char
 = Int -> (Bool -> Char)
```

Function application associates to the left:

```
f x y = (f x) y
```

The list type `[a]`

`[t]` is the type of (singly-linked) lists `[x1,x2,...,xn]` with elements of type `t`

Question. What is the type of the following lists?

- `[True,False] ::` ???
- `[1,2,3] ::` ???
- `[[1],[1,2],[1,2,3]] ::` ???

- `['a','b','c'] ::` ???

The list type `[a]`

`[t]` is the type of (singly-linked) lists `[x1,x2,...,xn]` with elements of type `t`

Question. What is the type of the following lists?

- `[True,False] :: [Bool]`
- `[1,2,3] :: Num a => [a]`
- `[[1],[1,2],[1,2,3]] :: Num a => [[a]]`
- `['a','b','c'] :: String`

Strings as lists

The type of strings is defined as `type String = [Char]`.

We can use the two types interchangeably:

```
> ['a','b','c'] == "abc"
True
> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
```

Note. The module `Data.Text` has a more efficient representation of strings.

Quiz question

Question. Which one is a correct type of the list `["True", "False"]`?

1. `String`
2. `[Bool]`
3. `[[Char]]`
4. `([Char], [Char])`

Lists vs. tuples

Two differences between a list and a tuple:

- An element of type `[t]` can have any length, while the size of a tuple of type `(t1,t2,...,tn)` is fixed.
- All elements in a list must have the same type, while the elements of a tuple can have different types.

Ranges of values

The list of values from `i` to `j` is written `[i..j]`.

Try to evaluate these ranges in GHCi:

- `[1..10]`
- `[1,3..10]`
- `[42..42]`
- `[10..1]`
- `[10,8..1]`

Ranges of values

The list of values from `i` to `j` is written `[i..j]`.

Try to evaluate these ranges in GHCi:

- `[1..10]` `= [1,2,3,4,5,6,7,8,9,10]`
- `[1,3..10]` `= [1,3,5,7,9]`
- `[42..42]` `= [42]`
- `[10..1]` `= []`
- `[10,8..1]` `= [10,8,6,4,2]`

List comprehensions

List comprehensions

We can construct new lists using a list comprehension:

```
[ x*x | x <- [1..5] ]
```

List comprehensions

We can construct new lists using a list comprehension:

```
[ x*x | x <- [1..5] ]
 = [1,4,9,16,25]
```

List comprehensions

We can construct new lists using a list comprehension:

```
[ x*x | x <- [1..5] ]
 = [1,4,9,16,25]
```

The part `x <- [1..5]` is called a generator.

(Compare with set comprehensions from mathematics: $\{x^2 \mid x \in \{1 \ldots 5\}\}$)

Using multiple generators (1/2)

We can use more than one generator:

```
[ x*y | x<-[1,2,3], y<-[10,100] ]
```

```
[ x*y | y<-[10,100], x<-[1,2,3] ]
```

Using multiple generators (1/2)

We can use more than one generator:

```
[ x*y | x<-[1,2,3], y<-[10,100] ]
 = [10,100,20,200,30,300]
```

```
[ x*y | y<-[10,100], x<-[1,2,3] ]
 = [10,20,30,100,200,300]
```

The order of the generators matters!

Using multiple generators (2/2)

Later generators can depend on previous ones:

```
[ 10*x+y | x<-[1..3], y<-[x..3] ]
```

Using multiple generators (2/2)

Later generators can depend on previous ones:

```
[ 10*x+y | x<-[1..3], y<-[x..3] ]
 = [11,12,13,22,23,33]
```

Filtering lists

We can select only elements that satisfy a boolean predicate:

```
[ x | x <- [1..10] , even x ]
```

Filtering lists

We can select only elements that satisfy a boolean predicate:

```
[ x | x <- [1..10] , even x ]
  = [2,4,6,8,10]
```

The predicate `even x` is called a guard.

Note: a guard must be of type `Bool`.

List comprehensions as a general control structure

Haskell does not have built-in control (such as `for` or `while`). Instead, we can use lists to define iterative algorithms:

Imperative code:

```
int result = 0;
int i = 0;
while (i<=10) {
  result += i*i;
  i += 1;
}
```

Haskell equivalent:

```
result =
  sum [ i*i
      | i <- [1..10]
      ]
```

Two exercises

Using list comprehensions,...

- define `addMod3Is2 :: [Int] -> [Int]` that keeps only values equal to 2 modulo 3 (2, 5, 8, etc.), and adds 3 to each.

```
addMod3Is2 [2, 3, 4, 8] = [5, 11]
```

- define `concat :: [[Int]] -> [Int]` that 'flattens' a list of lists into a single list.

```
concat [[1],[2,3],[]] = [1,2,3]
```

What's next?

Next lecture: Defining and testing functions

To do:

- Get the book
- Read the book:
  - Today: 1.1-1.5, 2.1-2.5, 3.1-3.6, 5.1-5.2
  - Next lecture: 3.7-3.9, 4.1-4.4, 6.1-6.6 + QuickCheck notes
- Install Stack and GHC
- Start with the exercises on WebLab