Functional Programming for Logicians - Lecture 1

Functions, Lists, Types

Malvin Gattinger

4 June 2018

```
module L1 where
```

Introduction

Who is who

Course website: https://malv.in/2018/funcproglog/

Malvin Gattinger

- Master of Logic 2012–2014
- PhD at ILLC 2014–2018
- cycling and dark chocolate

Jana Wagemaker

- Master of Logic 2015–2017
- PhD at CWI 2018–
- Beyoncé and horses

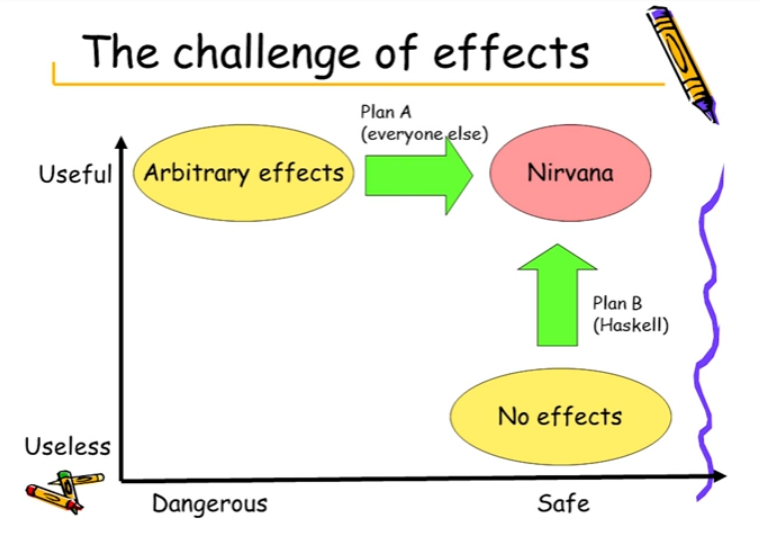
Functional Programming

- the main operation is function application
- describe *what*, not *how* it should be computed
- a program is a list of definitions of functions

Haskell

- lambda calculus meets category theory
- **typed**: every expression has a type fixed at compile time
- **lazy**: only compute what and when it is needed
- **pure**: functions have no side-effects
  - same input $\rightarrow$ same output

Why?



(Simon Peyton-Jones: *Escape from the ivory tower: the Haskell journey*)

Let's go

Calculating

We work in *ghci* for now, ~~an~~ the interactive compiler for Haskell.

```
λ> 7 + 8 * 9
79
λ> (7 + 8) * 9
135
λ> sum [1,6,10]
17
```

Functions

The definition

```
square x = x * x
```

gives us:

```
λ> square 9
81
λ> square 10
100
```

⇒ board exercise: Define `double`, `cube` and `plus`

Our first Type (Error)

```
λ> square "10"
<interactive>:3:8: error:
    • Couldn't match expected type ‘Integer’
                  with actual type ‘[Char]’
```

I lied before. 🤥

The definition of `square` we were actually using is this:

```haskell
square :: Integer -> Integer
square x = x * x
```

We read the :: double colon as “has the type”

In Haskell everything has a type!

⇒ board exercise: What are the types of `10`, `"10"`, `+`, `*` and `+5`?

Lists

```haskell
myList :: [Integer]
myList = [1,23,42,111,1988,10,29]

longList :: [Integer]
longList = [1..100]
```

```
λ> length myList
7
λ> length longList
100
λ> 1:3:myList
[1,3,1,23,42,111,1988,10,29]
λ> myList ++ [5,7] ++ myList
[1,23,42,111,1988,10,29,5,7,1,23,42,111,1988,10,29]
```

mapping over lists

```
λ> map square myList
[1,529,1764,12321,3952144,100,841]
λ> map square [1..4]
[1,4,9,16]
```

⇒ board exercise: What does `map` do? How can we define it?

⇒ Question: What is the type of `map`? Here? In general?

Hint: Pattern matching on `[]` and the `:` operator

Type Variables and Inference

```haskell
wordList :: [String]
wordList = ["beyonce","metallica","k3","anathema"]
```

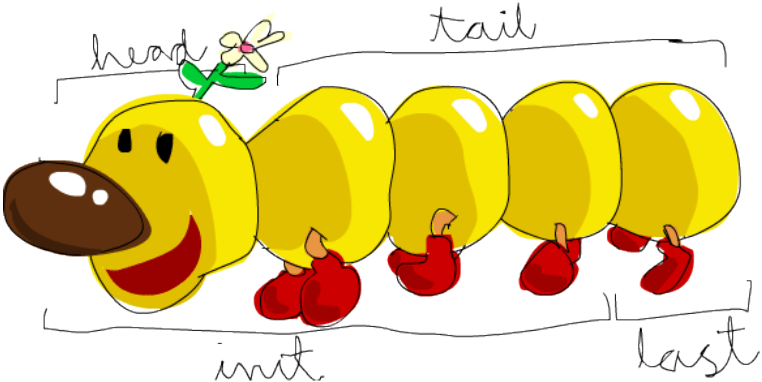
⇒ Why does `map square wordList` give an error? 🔥

Hint: Look at the error generated by this:

```
λ> import Data.Char
λ> :t toUpper
toUpper :: Char -> Char
λ> map toUpper wordList
...
```

The List Monster

⇒ board exercise: Define these four functions, start with the type!



picture from http://learnyouahaskell.com/starting-out/#an-intro-to-lists

Strings are lists of characters

In fact we have:

```
type String = [Char]
```

Example:

```
λ> "barbara" == ['b','a','r','b','a','r','a']
True
```

Strings are lists of characters

In fact we have:

```
type String = [Char]
```

Example:

```
λ> "barbara" == ['b','a','r','b','a','r','a']
True
```

Note the difference between `'` and `"`:

```
λ> :t 'a'
'a' :: Char
λ> :t "a"
"a" :: [Char]
```

$\Rightarrow$ Why does `'ab'` not make sense?

Mapping and Sorting Strings

```haskell
swab :: Char -> Char
swab 'a' = 'b'
swab 'b' = 'a'
swab c   = c
```

```
λ> map swab "abba"
"baab"
λ> map swab "barbara"
"abrabrb"

λ> import Data.List
λ> sort "hello"
"ehllo"
λ> sort "barbara"
"aaabbrr"
```

Infinite Lazy Lists

What happens here?

```
naturals :: [Integer]
naturals = [1..]
```

What happens if I evaluate `naturals` in ghci now?

Hint: Maybe I shouldn't 😉

Infinite Lazy Lists

What happens here?

```
naturals :: [Integer]
naturals = [1..]
```

What happens if I evaluate `naturals` in ghci now?

Hint: Maybe I shouldn't 😉

But we can ask for finite parts of it, lazily!

```
λ> take 11 naturals
[1,2,3,4,5,6,7,8,9,10,11]
λ> take 11 (map square naturals) -- not strict!
[1,4,9,16,25,36,49,64,81,100,121]
λ> map square (take 11 naturals)
[1,4,9,16,25,36,49,64,81,100,121]
```

⇒ Board exercise: Give a definition of `take`.

Recursion

```
sentence :: String
sentence = "Sentences can go " ++ onAndOn where
  onAndOn = "on and " ++ onAndOn
```

Try this out with `take 65 sentence` in ghci.

Type Hype

- `Integer`
- `Int`
- `[a]`
- `Char`
- `String = [Char]`

Type Hype

- `Integer`
- `Int`
- `[a]`
- `Char`
- `String = [Char]`

Tuples:

- `(a,b)`
- `(a,b,[c])`

Sum types:

- `Maybe a`
- `Either a b`
- `()`

Tuples

```
malvin, jana :: (String,Integer)
malvin = ("Malvin",1988)
jana = ("Jana",1993)
```

Tuples

```haskell
malvin, jana :: (String,Integer)
malvin = ("Malvin",1988)
jana = ("Jana",1993)
```

Can you guess what the following functions do?

```haskell
fst :: (a,b) -> a
snd :: (a,b) -> b
Data.Tuple.swap :: (a,b) -> (b,a)
```

Tuples

```haskell
malvin, jana :: (String,Integer)
malvin = ("Malvin",1988)
jana = ("Jana",1993)
```

Can you guess what the following functions do?

```haskell
fst :: (a,b) -> a
snd :: (a,b) -> b
Data.Tuple.swap :: (a,b) -> (b,a)
```

```
λ> fst malvin
"Malvin"
λ> snd malvin
1988
λ> swap jana
(1993,"Jana")
```

Lambdas

We write \ for λ to define an anonymous function:

```
λ> (\y -> y + 10) 100
110
λ> map (\x -> x + 10) [5..15]
[15,16,17,18,19,20,21,22,23,24,25]
```

$\Rightarrow$ board exercise: define `fst`, `snd` and `swap` with lambdas!

Function application and composition

```
people :: [(String,Integer)]
people = [jana,malvin]
```

```
λ> map (length . fst) people
[4,6]

λ> concat $ map fst people
"JanaMalvin"
λ> sum $ map snd people
3981
```

⇒ board exercises: 🤔

- What do `.` and `$` do?
- Why is `$` still useful?
- Why should we call this “point-free”?

List Comprehension

We can also build new lists using this notation:

```
threefolds :: [Integer]
threefolds = [ n | n <- [0..],  rem n 3 == 0 ]
```

Which is syntactic sugar for

```
filter (\n -> rem n 3 == 0) [0..]
```

The notation is close to *set* comprehension:

$$\{n \in \mathbb{N} \mid n \equiv 0 \mod 3\}$$

→ see also https://wiki.haskell.org/List_comprehension

Even more Lists

These are all the same:

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

```
[1,2,3,4,5,6,7,8,9,10]
```

```
1:2:3:4:5:6:7:8:9:10:[]
```

```
1:2:3:4:5:6:[7..10]
```

```
[ x | x <- [1..100],  x <= 10 ]
```

```
takeWhile (< 11) [1..]
```

What about this one? 🤔

```
filter (< 11) [1..]
```

Guards

Instead of  code like this ...

```haskell
magnitudeUgly :: Integer -> String
magnitudeUgly n = if n < 10
                    then "small"
                    else if n < 100
                           then "medium"
                           else "large"
```

... we can use *guards* like this:

```haskell
magnitude :: Integer -> String
magnitude n | n < 10    = "small"
            | n < 100   = "medium"
            | otherwise = "large"
```

⇒ What is the type of `otherwise` and what does it do? 

How to make a type

`type` defines types that are just abbreviations:

```
type Person = (String,Integer)
```

```
type Group = [People]
```

To create actually new types we use `data`:

```
data Animal = Cat | Horse
```

```
data Either a b = Left a | Right b
```

```
type Maybe a = Nothing | Just a
```

Note that this defines a new type and constructors at the same time!

Pattern matching

Each data type can be matched by *patterns*:

- Bool: `True`, `False`, `b`
- Lists: `[]`, `(x:xs)`, `(x:y:rest)`, ...
- Strings: `'h':'e':[]`, `"hello"`, ...
- Tuples: `(x,y)`
- Numbers: 1, 2, 3, 42, ...
- Maybe a: `(Just x)`, `Nothing`
- Either a b: `Left x`, `Right y`
- anything: `x`, `mySuperLongVarName`, `_`

Patterns can occur in two places.

First, as arguments of functions:

```haskell
isEmpty :: [a] -> Bool
isEmpty []    = True
isEmpty (_:_) = False
```

Second, in `case ... of ... -> ...` constructs.

Logic in Haskell

Propositional Logic

Being logicians, you can probably read this:

```
data Form = P Int | Neg Form | Conj Form Form
```

Formulas are defined by: $\varphi ::= p_n \mid \neg\varphi \mid \varphi \wedge \varphi$

```
type Assignment = Int -> Bool

satisfies :: Assignment -> Form -> Bool
satisfies v (P k)      = v k
satisfies v (Neg f)    = not (satisfies v f)
satisfies v (Conj f g) = satisfies v f && satisfies v g
```

Given an assignment $v \colon P \to \{\top, \bot\}$, we define:

- $v \vDash p_i \;:\Longleftrightarrow\; v(p_i)$
- $v \vDash \neg\varphi \;:\Longleftrightarrow\;$ not $v \vDash \varphi$
- $v \vDash \varphi \wedge \psi \;:\Longleftrightarrow\; v \vDash \varphi$ and $v \vDash \psi$

Examples

```
world :: Assignment
world 0 = True
world 1 = False
world 2 = True
world _ = False
```

```
λ> satisfies world (Neg . Neg $ P 2)
True
λ> satisfies world (Conj (Neg $ P 1) (P 0))
True
```

Preview

Actually, you want this:

```
data Form = P Int | Neg Form | Conj Form Form
  deriving (Eq,Ord,Show)
```

The `Eq`, `Ord` and `Show` are *type classes* which we will study tomorrow.

Practical Stuff

Organization

- First week:
  - Lectures Monday to Friday: 10:00-12:00
  - Exercise Sessions: Monday, Tuesday, Thursday and Friday: 14:00-16:00
- Second week: Start and present your own topic
  - Monday: start working on a topic
  - Friday: Presentations 10:00 to 12:00
- End of the month: submit report with code & documentation
  - Deadline for you: June 29
  - Deadline for us: July 6

See `https://malv.in/2018/funcproglog/`

Abbreviation Mania

- *GHC* is the Glasgow Haskell Compiler
- *GHCi* is the *interactive* interface of GHC
- `stack` is a build tool to simplify your life
- `cabal` is another tool and a package format
- *Hackage* is a public database of Haskell libraries
- *Stackage* provides stable snapshots, called *resolvers*.

We will use `stack 1.7.1`, `GHC 8.2.2` and resolver `lts-11.11`.

How to start

1. Install stack and make sure it is in your `PATH` variable.
2. Download `E1.lhs` and open a terminal where you saved it.
3. Run `stack exec ghci -- E1.lhs`
4. Edit the file.
5. Reload with `:r` and listen carefully if GHC shouts at you.
6. Try something out.
7. Go to 4.

🥗 🍝

See you again at 14:00 in F1.15.