Functional Programming for Logicians - Lecture 1

Functions, Lists, Types

Malvin Gattinger

12 January 2022

```
module L1 where
```

Introduction

Who is who

Who is who

You

- a **wide range** of programming experiences: nothing, Java, Python, Rust, Agda, Lean, Prolog, Lisp, C, C++, C#, Haskell, Dart, Vala, Kotlin, Mathematica, ...
- interests: Category Theory, Cognition, Dynamic Epistemic Logic, Inquisitive Semantics, Proof Theory, Recursion Theory, Truth Makers, ...

Who is who

You

- a **wide range** of programming experiences: nothing, Java, Python, Rust, Agda, Lean, Prolog, Lisp, C, C++, C#, Haskell, Dart, Vala, Kotlin, Mathematica, ...
- interests: Category Theory, Cognition, Dynamic Epistemic Logic, Inquisitive Semantics, Proof Theory, Recursion Theory, Truth Makers, ...

Malvin

- 2012–2014 MoL
- 2014–2018 PhD at ILLC
- 2018–2021 PostDoc in Groningen
- 2021– assistant prof at ILLC

Functional Programming

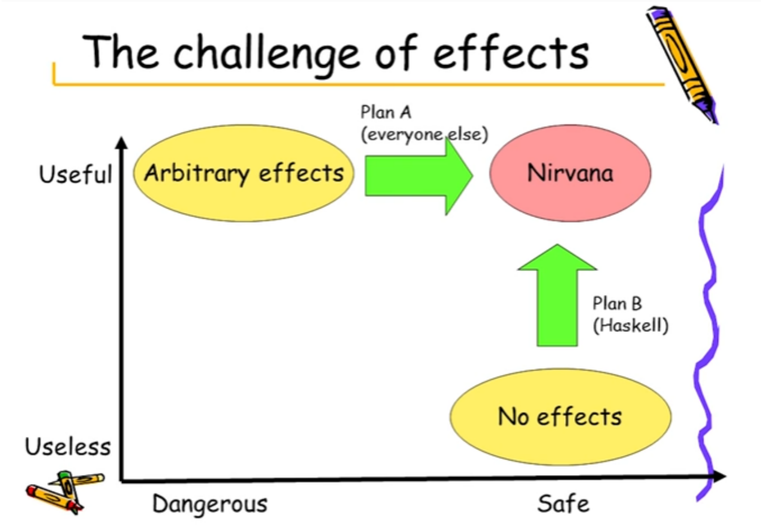
- the main operation is function application
- describe *what*, not *how* it should be computed
- a program is a list of definitions of functions

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?

Why?



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

Let's go

Calculating

We work in *ghc**i*** for now, the *interactive* compiler.

```
λ> 7 + 8 * 9
79

λ> (7 + 8) * 9
135

λ> sum [1,6,10]
17
```

Functions

Create a file `example.hs` which contains this:

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

Functions

Create a file `example.hs` which contains this:

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

Now we can run `ghci example.hs` and use this function!

Functions

Create a file `example.hs` which contains this:

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

Now we can run `ghci example.hs` and use this function!

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

Functions

Create a file `example.hs` which contains this:

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

Now we can run `ghci example.hs` and use this function!

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

⇒ How can we define `double`, `cube` and `plus`? 🤔

Our first Type (Error)

```
λ> square 10
100
λ> square "10"
<interactive>:3:8: error:
    • Couldn't match expected type ‘Integer’
                  with actual type ‘[Char]’
```

Our first Type (Error)

```
λ> square 10
100
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
```

Our first Type (Error)

```
λ> square 10
100
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!

Our first Type (Error)

```
λ> square 10
100
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!

⇒ 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]

λ> map (*5) [1,2,3,5]
[5,10,15,25]
```

mapping over lists

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

λ> map square [1..4]
[1,4,9,16]

λ> map (*5) [1,2,3,5]
[5,10,15,25]
```

⇒ What does `map` do? 🤔

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

mapping over lists

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

λ> map square [1..4]
[1,4,9,16]

λ> map (*5) [1,2,3,5]
[5,10,15,25]
```

⇒ What does `map` do?

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

How can we define `map`?

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

Type Variables and Inference

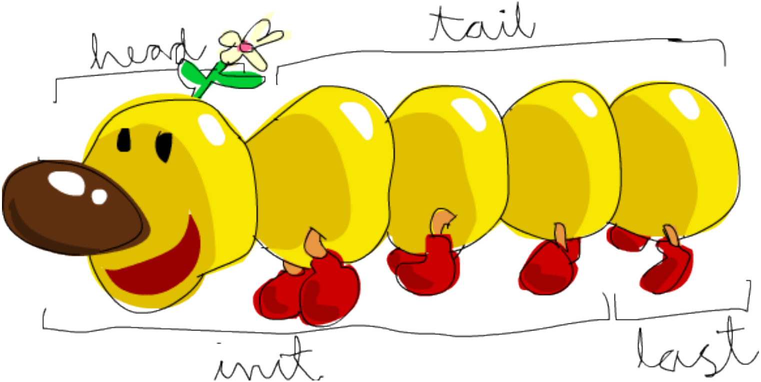
```
wordList :: [String]
wordList = ["beyonce","metallica","k3","anathema"]
```

⇒ 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



⇒ 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]
```

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]
```

⇒ Why does `'ab'` not make sense? 🤔

Mapping and Sorting Strings

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

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"
```

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?

```haskell
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> map square (take 11 naturals)
[1,4,9,16,25,36,49,64,81,100,121]
```

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> map square (take 11 naturals)
[1,4,9,16,25,36,49,64,81,100,121]

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

⇒ exercise: Give a definition of `take`.

Recursion

```haskell
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 (aka products):

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

Sum types:

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

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]
```

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]
```

$\Rightarrow$ How can we 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
```

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
```

⇒ Questions 🤔

- What do `.` and `$` do?
- Why is `$` still useful?
- Why should we call `(length . fst)` "point-free"?

List Comprehension

We can also build new lists using this notation:

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

The notation is close to *set* comprehension:

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

List Comprehension

We can also build new lists using this notation:

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

The notation is close to *set* comprehension:

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

An equivalent way to define the above:

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

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..]
```

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..]
```

But what about this one?

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

Is it the same value? Is it the same program? 🤔

Guards

Instead of 🍝 code like this . . .

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

Guards

Instead of 🍝 code like this ...

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

... we usually prefer *guards* like this:

```
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 = [Person]
```

How to make a type

`type` defines types that are *just abbreviations*:

```
type Person = (String,Integer)
type Group = [Person]
```

To create actually new types we use `data`:

```
data Animal = Cat | Horse | Koala
data MyEither a b = MyLeft a | MyRight b
data MyMaybe a = MyNothing | MyJust a
```

How to make a type

`type` defines types that are *just abbreviations*:

```
type Person = (String,Integer)
type Group = [Person]
```

To create actually new types we use `data`:

```
data Animal = Cat | Horse | Koala
data MyEither a b = MyLeft a | MyRight b
data MyMaybe a = MyNothing | MyJust a
```

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: `0`, `1`, `2`, `3`, `42`, ...
- Maybe a: `(Just x)`, `Nothing`
- Either a b: `Left x`, `Right y`
- anything: `x`, `mySuperLongVarName`, `_`

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: `0`, `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:

- as arguments of functions:

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

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

Logic in Haskell

Propositional Logic

Propositional Logic formulas are defined by: $\varphi ::= p_n \mid \neg\varphi \mid \varphi \wedge \varphi$

In Haskell:

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

Propositional Logic

Propositional Logic formulas are defined by: $\varphi ::= p_n \mid \neg\varphi \mid \varphi \wedge \varphi$

In Haskell:

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

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 $v \vDash \psi$

```
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
```

Examples

We define an assignment:

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

Examples

We define an assignment:

```
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)
```

`Eq`, `Ord` and `Show` are *type classes*, a topic for tomorrow.

Practical Stuff

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*.
- *VS Code* is a common and beginner-friendly editor.

Organization

Course website: https://malv.in/2022/funcproglog

Lectures on Zoom

Exercise Sessions on gather.town

$\Rightarrow$ Links are in the first email!

Organization

Course website: https://malv.in/2022/funcproglog

Lectures on Zoom

Exercise Sessions on gather.town

⇒ Links are in the first email!

Other useful tips for all-online:

- Try out screen-sharing in gather.town
- If you want to work live in pairs, use a collaborative editor!
  - EtherPad (plain text) on gather.town tables!
  - "Live Share" extension for VS Code
  - https://replit.com/

Literate Haskell

You can download the lecture and exercises as `.lhs` (or `.hs`) files.

This stands for "Literate Haskell" and is a way to combine programs and documentation or longer comments in one file.

In `.lhs` files the actual Haskell code has to be

- indented with with > (Markdown style) or
- between `\begin{code}` ... `\end{code}` (LaTeXstyle)

How to start

0. See instructions in first email to install Haskell and VS Code.
1. Download `E1.lhs` and open a terminal where you saved it.
2. Run `ghci E1.lhs` (or `stack ghci E1.lhs`).
3. Edit the file in VS code.
4. Reload with `:r` and read carefully what GHC tells you.
5. Try out all the things!
6. Go to 3.

🥗 🍝

See you again at 13:00 in gather.town.