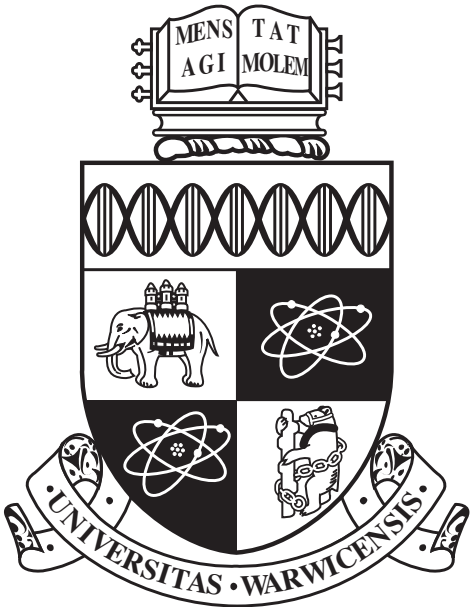
University of Warwick
Department of Computer Science

CS141

Functional Programming



Cem Yilmaz
January 30, 2022

Contents

1 What is Functional Programming

1.1 History

In 1928, David Hilbert posed the question "Given any true mathematical statement, is there an algorithm for verifying that it is true?". This is now known as the "Entscheidungsproblem", that is, the decision problem. E.g.

$$1 = 1$$
$$2 = 2$$
$$2 < 3$$
$$1 < 3$$

In 1931, Kurt Gödel came up with the paradox with the statement "This statement is not provable". If we assume not provable, there is a double negation which implies it is provable. This contradicts the assumption. If we assume true, we have a true statement that is not provable. This is called the incompleteness theorem. This answered the question that mathematics cannot answer every statement.

In 1936, Alonzo Church came up with $\lambda-$calculus as a system for describing algorithms. Kurt Gödel then believed that he can do better. He believed that some algorithms would not be able to be described with $\lambda-$calculus. Kurt then came up with a system of recursive functions. Alonzo Church then claimed that any algorithm that can be described using recursive functions can also be described using $\lambda-$calculus.

Alan Turing then came along and then came up with his own system of describing algorithms utilising Turing machines. However, he also showed that anything described using Turing machine could be described with $\lambda-$calculus. However, notice that despite these being different systems, they were also equivalent in describing algorithms.

1.2 Today

Today, programming as you know it, for example:

$$\prod_{i=1}^{4} = 1 \times 2 \times 3 \times 4$$

If we wanted to turn this to the roughly equivalent program in a language such as Java or C

```
int x = 1;
for (int = 1; i <= 4; i++) {
  x *= i;
}
```

Listing 1: Product in Java

Table 1: Table of results

Variable	Value
x	1
x	2
x	6
x	24

However, in a function language, we can express it as

```
product[1..4]
```

Listing 2: Product in Haskell

That is, product is a function. You can also expand this to be

```
product[1,2,3,4]
1 * product[2,3,4]
1 * 2 * product[3,4]
1 * 2 * 3 * product[4]
1 * 2 * 3 * 4
24
```

Listing 3: Expanded Product

The definition of product function is rather simple:

```
product [n] = n -- Described in terms of 2 equations. First it checks if there a single item
    and returns it
product(n:ns) = n * product ns -- Takes the the first item in the list and keeps the rest
```

Listing 4: Product function

Let us now compare imperative programming with functional

Imperative	Functional
Mutation of state	Reduction of expression
Tell the computer how you want to do something	Tell the computer what you want to compute and let it work out how to do it
Statements executed in order specified	Sub-expressions can often be evaluated in an arbitrary order
Loops	Recursion

1.3 Programming Paradigms

In history, there used to be a clear distinction between programming paradigms. However, today, this has changed. That is,

- Java / C are now multi-paradigm: They're imperative, object-oriented, functional, etc.
- Python, JavaScript, C++ have similarly multi-paradigm.

As such, learning functional programming will be useful as paradigms have blended together.

1.4 What Haskell is good for

1.4.1 Web Services

Furthermore, a particularly nice application to functional programming is that they're good at web services.

- Lots of cool frameworks for developing web applications
- Easy to embed domain-specific languages for routing, templates, etc.
- Servant: describe web service as a type, automatically generate client programs
- One of the coursework assignments uses a web service written in Haskell to provide a browser-based interface

1.4.2 Domain-specific language

Domain-specific languages are also a thing. For example, you can write music in a Haskell library to describe music.

```
import Mezoo

v1 = d qn :|: g qn :|: fs qn :|: g en :|: a en :|: bf qn :|: a qn :|: g hn
v2 = d qn :|: ef qn :|: d qn :|: bf_ en :|: a_ en :|: b_ qn :|: a_ qn :|: g_ hn

main = playLive (v1 :-: v2)
```

Listing 5: Music in Haskell

What's cooler is that if we try to compile, we would get an error that the composition is not harmonic, that is, if it does not sound good, it does not compile. In particular,

- Major sevenths are not permitted in harmony: Bb and B_
- Direction motion in a perfect octave is forbidden: Bb and B_, then A and A_
- Parallel octaves are forbidden: A and A_, then G and G_

1.4.3 Games

For example, the game magic cookies utilises functional reactive programming to describe its game logic. It is the same code across different platforms. It is also good for time-travel debugging.

1.4.4 System Software

- XMonad: Window manager
- OS: Mirage (OCaml), House (Haskell) and more

2 Basics of Haskell

2.1 Modules

Haskell code is separated into files called modules. A module file always begin with

```
20 module Whatever where
21 ...
```

Listing 6: Module

Note that the filename ends with $.hs$, and is the same as the module name.

2.2 Imports

Similarly, you can import nearby modules. For example, for the file $foo.hs$:

```
22 module foo where
23 double x = x * 2
```

Listing 7: Import

And then in another file,

```
24 module bar where
25 import Foo
26 quadruple x = double (double x)
```

Listing 8: Import of foo

In the same way, libraries are imported as they are just functions. For example,

```
27 module Whatever where
28 import Data.Char
29 swapCase c =  if isUpper c
30               then toLower c
31               else toUpper c
```

Listing 9: Library import

2.3 Definitions

A Haskell file is a series of definitions. For example, for the code

```
32 module Defn where
33 triangle n = sum [1..n]
```

Listing 10: Defn.hs

On the LHS of $=$ is a name, with zero or more arguments. On the RHS is an expression which utilises the arguments. Note that Haskell does not utilise parentheses. Arguments are separated by whitespace.

2.4 Lambda Notation and Mapping notations

Recall that in lambda calculus we denote inputs using lambdas, that is

$$\lambda xy.x \times y$$
$$=\lambda x \lambda y.x \times y$$

Similarly, in Haskell, the lambda notation is blended with the mapping arrow. That is, for example, we can define our function $multiply$ in following ways:

```
34 import Multiply where
35 multiply x y = x * y
36 multiply = \x y -> x * y
37 multiply x = \y -> x * y
38 multiply = \x -> \y -> x * y
```

Listing 11: Multiply.hs

These all are the same thing.

2.5 Partial Functions

It is possible to provide a singular argument into a function that takes in multiple arguments. For example, for our code above, if declared $multiply5$, then we would replace x with 5. Note that it is x in particular because it is the variable that is declared first.

```
39 multiply = \x y -> y * x
40 multiply 5 --(\y -> y * 5)
```

Listing 12: Multiply 5

2.6 Smooth Operator

Other than declaring functions, it is also possible to declare operators in a similar fashion. Consider the following code:

```
41 x ^^^ y = max x y
```

Listing 13: Operator

One can spot the similarities. However, note that operator names are made up of symbols rather than letters. Similarly to partial functions, there are also partially applied operators. For example,

```
42 plusFive = (+ 5)
```

Listing 14: Partially applied operator

This would add a 5 to any other operator.

2.7 Difference between a function and an operator

Functions are a prefix, whilst operators go between arguments (infix). It is possible to treat a function like an operator. Consider the following code:

```
GHCi > 5 `max` 6
6

GHCI> (*) 5 6
30
```

Listing 15: Function to operator

For example, the max function is enclosed within “ which allows us to use it like an operator.The latter operation is another alternative way of writing it.

2.8 Pattern Matching

Consider the following definition of factorial of a variable x :

```
fac x = if x == 0
           then 1
           else x * f (x - 1)
```

Listing 16: factorial

This can be rewritten as

```
fac x = case x of
    0 -> 1
    n -> n * f (n - 1)
```

Listing 17: factorial redefined

You can also express it in top-level patterns. That is, we pattern match our arguments directly

```
fac 0 = 1
fac x = x * f (x - 1)
```

Listing 18: factorial pattern matched

Often, Haskell code is written like this Lastly, you can also introduce "guards" for the code. If we need to check some predicate of the input and not a specific pattern

```
f x
      | x == 0       = 1
      | otherwise    = x * f(x-1)
```

Listing 19: factorial guard

3 Data Types

3.1 Compiler

The GHCi is an interpreter and runs a REPL loop, that is

- Read
- Evaluate
- Print

- Loop

Cabal is a program that manages Haskell packages. The Haskell package repository, Hackage, contains a bunch of packages. It downloads the package and then connects it to GHC. However, instead, we will be using Stack. It is a wrapper around cabal and ghc. It makes them both easier to use and helps with sharing programs across different systems.

3.1.1 Compiling

Definition 3.1. Compiling

Compiling is the process of turning raw source code into a format the computer can run.

In Haskell, GHC has three main compilation stages:

$$\text{(Raw source)} \implies \text{Parser} \implies \text{Type Checker} \implies \text{Code Generator} \implies \text{(Binary code)}$$

3.1.2 Types

In Java, we have types e.g.

```
59 int x = 5;
```

Listing 20: Java data type

In Haskell, every expression has a type. For example,

```
60 someBoolean = True && False || True
```

Listing 21: Boolean Type

In this case, someBoolean has the type true.

```
61 five = min 5 6
```

Listing 22: Integer

Five has type Integer. Integer does not have an upper bound unlike in other languages.

```
62 string = "Hello, world!"
```

Listing 23: String

In this case, string has the type string. You can also use GHCi to find the type that a function has. You can

3.1.3 Type Declarations

You can declare the type of a function just above it using

```
63 five :: Integer
64 five = min 5 6
```

Listing 24: Declaration of Data

:: is a special piece of notation that declares the data type of a function. However, if every expression has a type, how did we get away without writing them? The answer is, Haskell figures out what types to use. This is called *type inference*. In fact, in Haskell, even if we declare the type, Haskell checks that for us regardless. And if it checks and it does not line up, we get a type error. The benefit of declaring types is to pinpoint any possible errors that occur when compiling. Some errors include but are not limited to:

- Giving a function too many arguments

- Forgetting arguments
- Trying to use a value where it does not make sense

Types only exist at compile time. The type checker has guaranteed that all types Lin up, so it does not need to keep that information any more, meaning that we cannot check the type of a value at run time. It is also important to note that Haskell is a *strongly typed* language which means that the types are correct before it runs the program. For example, in Python, it is weakly typed and thus it make take you hours to find the error.

3.1.4 Function types

In mathematics, the succ function is a function s.t.

$$succ(n) = n + 1$$

We might say that $succ$ is a function from integers to integers, otherwise

$$succ : \mathbb{Z} \to \mathbb{Z}$$

In Haskell, it is written the same.

```
65 succ :: Integer -> Integer
66 succ n = n + 1
```

Listing 25: Mapping in Haskell

3.1.5 Polymorphism

In Haskell, we need to find a method to make sure that we can parse a function with a different types. For example,

```
67 id x = x
```

Listing 26: Identity Function

To give a general type here, we need to use a type variable. We write these with lowercase letters. In particular,

```
68 id :: a -> a
69 id x = x
```

Listing 27: Example

The compiler will specialise the type for us when we use the function. We can also ignore arguments using

```
70 ignoreSecondArgument x y = x
```

Listing 28: Ignore argument

This function takes two arguments and throws away the second one. Since it doesn't use any specific details about either argument, we can treat both arguments as being polymorphic.

4 Tuples and Lists

4.1 Tuples

Definition 4.1. Tuple

A tuple is a sequence of known, finite length. For example,

$$(1, 3) \in \mathbb{Z}^2$$

And in natural language, we know that this equates to "the pair $(1, 3)$ is an element of the set of pairs of integers". Indeed, this is not only restricted to pairs, but we could also have

$$(5, True, 0.2) \in \mathbb{Z} \times \mathbb{B} \times \mathbb{Q}$$

The number of elements in a tuple is called the dimension.

In Haskell, the tuples are represented in a very similar manner. For example,

```
71 (5, "Hello")
```

Listing 29: Tuple

This is the definition of a tuple in Haskell, and this is the only syntax that you require.

4.2 Types of Tuples

To declare the types of elements in a tuple, you would similarly declare it in a tuple as well. That is, for example

```
72 (False, 5) :: (Bool, Int)
```

Listing 30: Tuple type

With this you can also write functions that can take tuples as an argument. For example,

```
73 combine :: (Int, Int) -> Int
74 combine (x,y) = x + y
```

Listing 31: Tuple input

And indeed functions that also return tuples. We can similarly also use pattern matching:

```
75 combine p = case p of
76   (x,y) -> x + +y
```

Listing 32: Pattern Matching

Pattern matching allows us to inspect the values in a complex type based on its constructors. For tuples, (,) is the constructor. With pattern matching we destructuring, unpacking the pair to get at the values inside of it.

4.3 Polymorphic Functions

We can indeed write polymorphic functions over tuples, for example

```
77 fst :: (a,b) -> a
78 fst (x,y) = x
```

Listing 33: Polymorphic tuple

4.4 Currying

We know we can get carry-on functions for things such as

```
f :: Int -> Int -> Int
f :: Int -> (Int -> Int)
```

Listing 34: Currying

These two are equivalent. That is, the if we were to put in one argument in a function above, we would get another function that returns int -> int. The process of taking a function that takes several arguments at once is called currying. However, sometimes we want to write functions over pairs in terms of functions over two arguments.

```
curry :: ((a, b) -> c) -> ( a -> b -> c )
uncurry :: (a -> b -> c) -> ((a, b) -> c)
```

Listing 35: Curry and Uncurry

These functions would allow us to convert how many inputs we take.

4.5 Lists

Lists in Haskell have two constructors

```
[] :: [a]
```

Listing 36: Lists

The first constructs an empty list. Notice its type is polymorphic - it could be an empty list of any type. Lists in Haskell have two constructors. For literals,

```
==> [True, False, True]
```

Listing 37: Literals

Similar with tuples, we use pattern matching to restructure a list.

```
startsWithFive :: [int] -> Bool
startsWithFive [] = False
startsWithFive (x:xs) = x == 5
```

Listing 38: Destructure

We can also syntactic sugar this by

```
startsWithFive :: [Int] -> Bool
startsWithFive (5:xs) = True
startsWithFive _ = False
```

Listing 39: Nest Patterning

4.6 Head and Tail

We can get the head and tail of a list by using functions which are defined as

```
head :: [a] -> a
tail :: [a] -> [a]
```

Listing 40: Heads and Tail

However, we cannot get the first the head for an empty list. Furthermore, lists are NOT arrays. Operations on a list are too complex and require too many steps.

4.7 List Comprehensions

Haskell borrows comprehension syntax for its list. That is,

```
93 a = [ x^2 | x <- [1,2,3] ]
```

Listing 41: List Comprehension

We call $x <- [1, 2, 3]$ a generator. It binds each of the values in turn so they can be used. x^2 is an expression. In other words, the code expresses

$$a = \{x^2 | x \in \{1, 2, 3\}\}$$

In particular, for a list such as

```
94 [ (x,y) | x <- [0..3], y <- [0..x] ]
95 // The output is
96 [ (0,0)
97 , (1,0), (1,1)
98 , (2,0), (2,1), (2,2)
99 , (3,0), (3,1), (3,2), (3,3)
100 ]
```

Listing 42: Multiple Generators

We can also set up rules for our generators. For example, for the equations

$$Evens = \{x | x \in \mathbb{N}, x \text{ is even}\}$$

In Haskell,

```
101 evens = [ x | x <- [0..], even x]
```

Listing 43: Even

We call this boolean predicate a guard.

5 Map and Filter

5.1 Filtering

Suppose our previous definition for square numbers. However, how can we get even square numbers? Our first option is to wrap it with another list comprehension:

```
102 squares = [ x^2 | x <- [1..] ]
103 evenSquares = [ s | s <- squares, even s]
```

Listing 44: filtering

However, we can use a filter function as well

```
104 filter :: (a -> Bool) -> [a] -> [a]
105 evenSquares = filter (\s -> even s) squares --which is the same as
106 evenSquares = filter even squares
```

Listing 45: filter

This is called η reduction.

5.2 Mapping

Filter works well until we require to apply a function to a whole list. The map function is defined with the following arguments:

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

Listing 46: Mapping

For example,

```
GHCi> map succ [1,2,3]
[2,3,4]
```

Listing 47: Mapping succ

5.3 Ranges

Lists can be used to write a range of values without having to explicitly type them. The syntax for this is as follows:

```
[x .. z]
```

Listing 48: Range

where x is the start of the range and z is the end of the range. It is also possible to have second element and this can set the step amount. For example

```
[2,4 .. 16]
```

Listing 49: Step Range

This would include all even numbers from and including 2 to 16. However note that ranges must always proceed in an arithmetic progression. However, note that the range syntax is a little broken for floating point values.

6 Type Classes

Type classes allow us to define specific rules for functions whose we don't know the type.

6.1 Eq class

```
(==) Eq a where
  (==) :: a -> a -> Bool
```

Listing 50: Eq Class

The first line defines the name of the type of class (Eq) and a type variable (a) to use in the definitions. After that comes a series of function type definitions, without implementations. The type class declaration is a specification that must be implemented for each type that wants to be part of the Eq type class. This is why if you type

```
:t 5
-- The output is --
5 :: Num p => p
```

Listing 51: Class

And at that point, the program itself still has not specified whether its a float, int, etc. but has defined it as a num type clas.