Functional Programming

Lecture 1 — Introduction

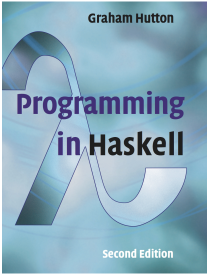
Pedro Vasconcelos
DCC/FCUP

2025

Contents and objetives

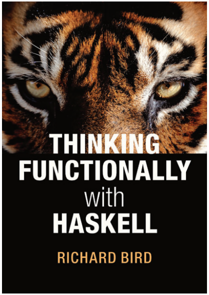
- Introduction to functional programming using the Haskell language
- By the end of the course, you should be able to:
  - understand the distinction between values, expressions and types
  - define functions using equations and pattern matching
  - define list processing functions by composition of Prelude functions and list comprehensions
  - define recursive data types for trees
  - use higher-order functions and lazy evaluation for better modularity
  - define programs that perform I/O using do-notation
  - define parsers for recursive data structures using combinators

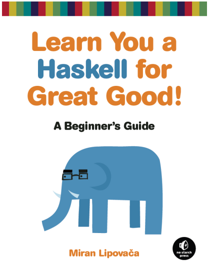
Recommend bibliography



- *Programming in Haskell*, 2nd edition, Graham Hutton, Cambridge University Press, 2016.

Secondary bibliography





- *Thinking functionally with Haskell*, Richard Bird. Cambridge University Press, 2015.
- *Learn you a Haskell for great good!*, Miran Lipovača. `http://learnyouahaskell.com/`

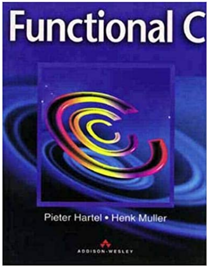
What is functional programming?

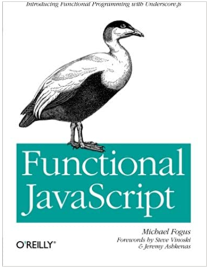
- A *programming paradigm* i.e. a philosophy on how to write programs
- Programs in C or Java are *imperative*: a program is a sequence of commands that modify variables in memory
- In the *functional paradigm* the program is a set of functions (in the mathematical sense)
- In a purely functional program we *never* mutate variables, we only apply functions
- A functional program is modeled as a function

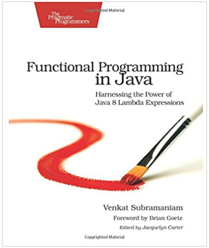
  $$\mathit{input} \longrightarrow \mathit{output}$$

  expressed by composition of simpler functions

Functional languages







- We can program in a functional style in almost any language
- But some languages explicitly *encourage* or even *enforce* the functional style
- Examples: Scheme, ML, OCaml, F#, Scala, Haskell

Example

To exemplify the imperative and functional paradigms, let us see two short programs for computing

$$1^2 + 2^2 + 3^2 + \cdots + 10^2$$

in Python and Haskell.

Sum squares — imperative version

```
# This is Python
total = 0
for i in range(1, 11):
     total = total + i*i
print(total)
```

- The program is written as a sequence of commands
- The sum is computed by mutating the state of variables over time
- We can visualize this using `https://pythontutor.com/`

Sum squares — functional version

```
-- This is Haskell
square x = x*x
main = print (sum (map square [1..10]))
```

- `[1..10]` is the sequence of integers from 1 to 10
- `square` is the function that computes the square of a number
- `map square` computes the square for every value in a sequence
- `sum` sums a sequence of values
- `print` prints the result

Step-by-step reduction

The execution of a functional program is done by step-by-step reduction, until we get to a result than cannot be further simplified.

```
   sum (map square [1..10])
=
   sum [1*1, 2*2, 3*3, 4*4, 5*4, 6*6, 7*7, 8*8,
        9*9, 10*10]
=
   1*1 + 2*2 + 3*3 + 4*4 + 5*5 + 6*6 + 7*7 + 8*8 +
   9*9 + 10*10
=
   1 + 4 + 9 + 16 + 25 + 36 + 49 + 64 + 81 + 100
=
   385
```

Step-by-step reduction (cont.)

Unlike the imperative program, we can perform operations by another order and still get the same result.

```
  sum (map square [1..10])
=
  sum [1*1, 2*2, 3*3, 4*4, 5*5, 6*6, 7*7, 8*8,
       9*9, 10*10]
=
  sum [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
=
  1 + 4 + 9 + 16 + 25 + 36 + 49 + 64 + 81 + 100
=
  385
```

You can try step-by-step reductions in my *Haskelite* interpreter: `https://pbv.github.io/haskelite`.

Why learn functional programming?

Thinking at a higher abstraction level

- More concise programs
- Closer to a mathematical specification
- More focus on understanding the problem and less on debugging
- Will make you a better programmer in any other language

> *A language that doesn't affect the way you think about programming is not worth knowing.*

Alan Perlis (1922–1990)

Why learn functional programming? (cont.)

More modularity

- decomposing problems into small, re-usable components

Correctness guarantees

- correctness proofs using mathematical techniques
- greater ease in doing automatic tests

Concurrency/parallelism

- greater freedom in changing the execution order without affecting results
- purely-functional data structures can be freely shared between threads

Disadvantages of functional programming

Greater distance from the hardware

- interpreters and compilers are more complex
- execution can be slower and/or require more space
- can be difficult to predict space/time execution costs
- some low-level programs require precise space/time control
- some data structures and algorithms are more efficient when implemented in an imperative way

A bit of history

1930s Alonzo Church develops the λ-calculus, a mathematical formalism for expressing computation using functions

1950s Inspired by the λ-calculus, John McCarthy develops LISP, one of the first high-level programming languages

1970s–1980s Robin Milner develops Standard ML, the first funcional language with *polymorphism* and *type inference*

1970s–1980s David Turner develops several languages that employ *lazy evaluation*, culminating in the commercial language Miranda

1987 An academic committee starts the development of Haskell, a standardized functional language employing lazy evaluation

2002 Publication of the *Haskell 98 report*

2010 Publication of the *Haskell 2010 report*

The Haskell language

`http://www.haskell.org`

- A general purpose, purely-functional language
- Named after the American logician Haskell B. Curry (1900–1982)
- Designed for teaching, research and also application development
- Result of over thirty years of active research and open-source contributions
- Some high-profile industrial usage in the last 25 years
- Principal implementation: the *Glasgow Haskell Compiler* (GHC)

Haskell in industry

NASA a domain-specific language for realtime systems
`https://copilot-language.github.io/`

QBayLogic design of FPGAs and ASICs
`https://qbaylogic.com/`

Standard Chartered Financial software
`https://serokell.io/blog/haskell-in-production-standard-chartered`

Meta Various internal tools and frameworks
`https://serokell.io/blog/haskell-in-production-meta`

Chordify Automatically convert any song into guitar chords
`https://serokell.io/blog/haskell-in-production-chordify`

Haskell in *open-source*

GHC the Haskell compiler is written in Haskell
`https://www.haskell.org/ghc/`

Shellcheck finds bugs in shell scripts
`https://www.shellcheck.net/`

Pandoc convert between various markup formats
`https://pandoc.org/`

Xmonad a tiling window manager
`https://xmonad.org/`

Codex my web system for programming exercises ☺
`https://github.com/pbv/codex`

Glasgow Haskell Compiler (GHC)

- A compiler from Haskell into native machine code
- Also includes an interpreter for interactive use (GHCi)
- Supports Haskell 2010 and many extensions
- Supports extensive code optimizations, foreign-function interface, build tools, profiling, etc.
- Available at `https://www.haskell.org/ghcup/`

First steps

Linux/MacOS/WSL2 execute the command `ghci`

```
$ ghci
GHCi, version 9.6.7: https://www.haskell.org/ghc/
:? for help
ghci>
```

Using the GHCi interpreter

The interpreter *reads* an expression, *evaluates* it, *prints* the result and repeats (*read-eval-print* loop).

```
ghci> 2+3*5
17
ghci> (2+3)*5
25
ghci> sqrt (3^2 + 4^2)
5.0
```

Operators and arithmetic functions

`+`	adition
`-`	subtraction
`*`	multiplication
`/`	fractional division
`^`	power (integer exponent)
`div`	quotient
`mod`	remainder
`sqrt`	square root
`==`	equals
`/=`	not equals
`<` `>` `<=` `>=`	comparisons

Syntax conventions

- Function arguments are separated by spaces
- Function application has higher precedence than any operator

Haskell	Usual math notation
`f x`	$f(x)$
`f (g x)`	$f(g(x))$
`f (g x) (h x)`	$f(g(x), h(x))$
`f x y + 1`	$f(x, y) + 1$
`f x (y+1)`	$f(x, y + 1)$
`sqrt x + 1`	$\sqrt{x} + 1$
`sqrt (x + 1)`	$\sqrt{x + 1}$

Syntax conventions (cont.)

- Any operator can be used as a binary function by writing in parenthesis
- Conversely, a function can be used as an operator by writting it between back-quotes (NOT single quotes!)

```
(+) x y      ≡  x+y
(*) y 2      ≡  y*2

x`mod`2      ≡  mod x 2
f x `div` n  ≡  div (f x) n
```

The Standard Prelude

The *Prelude* module contains many predefined functions:

- all operations on numbers and related functions;
- generic functions on lists;
- and many others: `https://www.haskell.org/onlinereport/haskell2010/haskellch9.html`

The Prelude is imported by default in the GHCi interpreter and in every Haskell module.

Some functions from the Prelude

```
ghci> head [1,2,3,4]
1
ghci> head "banana"
'b'
```

get the first element

```
ghci> tail [1,2,3,4]
[2,3,4]
ghci> tail "banana"
"anana"
```

remove the first element

```
ghci> length [1,2,3,4,5]
5
ghci> length "banana"
6
```

get the length

Some functions from the Prelude (cont.)

```
ghci> take 3 [1,2,3,4,5]
[1,2,3]
ghci> take 3 "banana"
"ban"
```

get a prefix

```
ghci> drop 3 [1,2,3,4,5]
[4,5]
ghci> drop 3 "banana"
"ana"
```

remove a prefix

```
ghci> [1,2,3] ++ [4,5]
[1,2,3,4,5]
ghci> "aba" ++ "cate"
"abacate"
```

concatenate

Some functions from the Prelude (cont.)

```
ghci> reverse [1,2,3,4,5]
[5,4,3,2,1]
ghci> reverse "abacate"
"etacaba"
```

invert the order

```
ghci> [1,2,3,4,5] !! 3
4
ghci> "abacate" !! 3
'c'
```

indexing (from 0)

```
ghci> sum [1,2,3,5]
11
```

sum all values

```
ghci> product [1,2,3,5]
30
```

product of all values

Define new functions

- We can define new functions in a text file
- Use your preferred code editor (e.g. Vim, Emacs, VS Code)
- Be sure to install the Haskell syntax highlighting mode
- Advanced IDE features like the “Haskell language server” are *not* necessary
- The filename should have the extension “`.hs`”

Creating a file with definitions

Listing 1: test.hs

```
double x = 2*x

quadruple x = double (double x)
```

Use the *:load* command to load a file in GHCi.

```
$ ghci
...
ghci> :load test.hs
[1 of 1] Compiling Main ( test.hs, interpreted )
Ok, modules loaded: Main.
```

Example

```
ghci> double 2
4
ghci> quadruple 2
8
ghci> take (quadruple 2) [1..100]
[1,2,3,4,5,6,7,8]
```

Modifying our file

Let us add new definitions and save the file again.

Listing 2: test.hs

```
factorial n = product [1..n]

average x y = (x+y)/2
```

We use *:load* again or just *:reload* to update GHCi.

```
ghci> :reload
ghci> factorial 10
3628800
ghci> average 2 3
2.5
```

Useful interpreter commands

`:load` *file*	load a file
`:reload`	reload changes
`:type` *expr*	show the type of an expression
`:help`	get some help
`:quit`	end the GHCi session

Short-hand notation:

`:l` instead of `:load`

`:r` instead of `:reload`

`:t` instead of `:type`

`:q` instead of `:quit`

Identifiers

The names of functions and variables must start by a lower case letter and may contain letters, digits, underscore (_) or single quotes.

```
fun1     x_2     y'     fooBar
```

The following *reserved words* cannot be used as identifiers:

```
case class data default deriving do else
if import in infix infixl infixr instance
let module newtype of then type where
```

Local names

We can introduce local names using `where`.

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

The indentation indicates the scope of declarations; we can also use explicit grouping using braces and semicolons:

```
a = b+c
  where {b = 1;
         c = 2}
d = a*2
```

Indentation

All definitions in a single scope should start in the same column.

```
a = 1
 b = 2
c = 3
```

WRONG

```
  a = 1
b   = 2
  c = 3
```

WRONG

```
a = 1
b = 2
c = 3
```

OK

The order among definitions is not important (these are equations, not assignments).

Comments

Simple start with -- and extend till the end of the line

Multi-line between {- and -}

```
-- Compute the factorial of an integer
factorial n = product [1..n]

-- Compute the average of two numbers
average x y = (x+y)/2

{-  The following definitions are commented out:
double x = x+x
square x = x*x
-}
```