Haskell and the power of functional programming

Mohabat Tarkeshian

MATH 9171L: MATHEMATICAL COMPUTATION

June 24, 2021

Overview

1. A little history
2. Applications in the “real” world
3. Why functional programming?
4. The basics
5. Lazy evaluation
6. Examples and demoing Haskell

A little history

- First Haskell Language Report: 1990
- Stable release: 2010
- Widely used in teaching, research, and industry
  - Annual research conference: ACM Haskell Symposium
- A statically typed functional programming language
- Lazy evaluation and typeclasses

Motivation: The "real"-world

- Game position optimization
- Document conversion (Pandoc)
- Extracting LaTeX code from a handdrawn symbol (Detexify)
- Extracting music chords (Chordify)
- Internal IT infrastructure (Google)
- Multicore parallelism (Intel)
- Secure contract signatures (Scrive)
- Blockchain and cryptocurrency (Cardano & Ada)
- Supply chain optimization (Target)
- Copilot project (NASA & Galois Inc.)
- Mobile electronic health records (Factis research)
- Building declarative animations (Reanimate)

A fun fact: Haskell is written in Haskell!

Functional vs imperative programming

Imperative programming

- Define a sequence of executable *tasks*
- Variables can change their state while executing functions
- Control flow structures for repeating some action several times
- *Sequential* thinking

Functional programming

- Define what things *are* - everything is encoded as a function
- Variables are *static*
- Functions do not have *'side-effects'*
  - But.. we can interact with the real-world using an I/O action
- Glue any number of functions and programs together: *modular* thinking

The basics

How are functions defined?

1. Indicate the input and output types (not strictly necessary)
2. Function name
3. A space
4. Input parameters
5. Output
    - Might include *pattern matching*

```
double :: Int -> Int
double n = 2 * n

factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n-1)
```

List comprehension and infinite lists: The way you want it to be

- Encode sets as we would mathematically

```
list1 :: [Int]
list1 = [2*x | x <- [1..10], 2*x >= 12]

removeLowercase :: String -> String
removeLowercase st = [ c | c <- st, c `elem` ['A'..'Z
    ']]
```

- Can define *infinite lists*

```
[1, 2..]
[2, 4..]
```

But how does this work?

Lazy evaluation

- Unless something is necessary, it is not evaluated
  - Will not compute every element of an infinite list to invoke a function that only requires a finite subset
- Avoids infinite recursion
- Efficiency - it's complicated
- Thinking mathematically in "thunks"

Examples using Emacs and `ghc`

- "Invalid" computations
- Infinite structures:
  - $\mathbb{N}$
  - Primes
  - The Fibonacci numbers
  - Cycles
- A tree as a functor

References

- Andrej Bauer
  Mathematics and computation: A blog about mathematics for computers
  Hask is not a category

- Haskell in industry
  Haskell Wiki

- John Hughes
  Why functional programming matters
  1990
  Research Topics in Functional Programming

- Miran Lipovača
  Learn you a Haskell for great good
  2011