

CS339: Abstractions and Paradigms for Programming

Lazy Typed Programming with Haskell

Manas Thakur
CSE, IIT Bombay



Autumn 2024

Case Study on Typed Functional Programming

- Recall this slide?

The APP Course

- Empowers us to **choose** the right PL for the task at hand;
- Makes **learning** a new PL much easier;
- Equips us in **designing** a new PL and its interpreter;
- Is huge **fun**!

Side learnings:

- Ability to comprehend large programs
- Language technologies that everyone uses
- Crux of **future** programming languages
- Different *right* ways of thinking about the same thing
- Motivation to learn the art of **cooking** (CS302) and **processing** (CS614, CS618, CS6004) the **food** that everyone (CSXYZ) eats



Manas Thakur

11



What's a function?

- A map from values in a domain D to values in a range R.

`f :: D -> R`

- Examples:

```
sin :: Float -> Float
age :: Person -> Int
add :: (Integer,Integer) -> Integer
```

- Applying a function:

$f(x)$
 $\sin \theta$ OR $\sin(\theta)$?
Haskell: `f x`



Function composition

`f :: Y -> Z`

`g :: X -> Y`

➤ What is `f . g`?

`f . g :: X -> Z`

`(f . g) x :: f (g x)`

➤ All this is Haskell code!





An advanced, purely functional programming language

Features

Statically typed

Every expression in Haskell has a type which is determined at compile time. All the types composed together by function application have to match up. If they don't, the program will be rejected by the compiler. Types become not only a form of guarantee, but a language for expressing the construction of programs.

Click to expand

Type inference

You don't have to explicitly write out every type in a Haskell program. Types will be inferred by unifying every type bidirectionally. However, you can write out types if you choose, or ask the compiler to write them for you for handy documentation.

Click to expand

Lazy

Functions don't evaluate their arguments. This means that programs can compose together very well, with the ability to write control constructs (such as if/else) just by writing normal functions. The purity of Haskell code makes it easy to fuse chains of functions together, allowing for performance benefits.

Click to expand

Declarative, statically typed code.

```
primes = filterPrime [2..] where
  filterPrime (p:xs) =
    p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

Purely functional

Every function in Haskell is a function in the mathematical sense (i.e., "pure"). Even side-effecting IO operations are but a description of what to do, produced by pure code. There are no statements or instructions, only expressions which cannot mutate variables (local or global) nor access state like time or random numbers.

Click to expand

haskell.org

Concurrent

Haskell lends itself well to concurrent programming due to its explicit handling of effects. Its flagship compiler, GHC, comes with a high-performance parallel garbage collector and light-weight concurrency library containing a number of useful concurrency primitives and abstractions.

Click to expand

Packages

Open source contribution to Haskell is very active with a wide range of packages available on the public package servers.

Click to expand



Common Words



```
the: 154
of : 50
a   : 18
and: 12
in  : 11
...
...
n entries
```

Type declaration

- What's the type of input?

- An integer

- A list of characters

Int, [Char]

- What's the type of output?

- List of characters again!

[Char]

- What's the type of commonWords?

`commonWords :: Int -> ([Char] -> [Char])`

Functions in Haskell are **curried**!



Possible algorithm

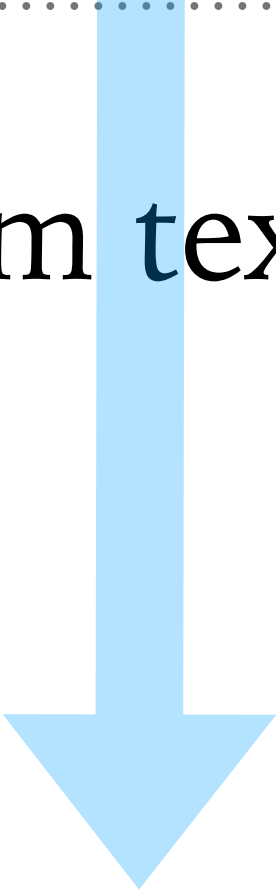
- Get words from text
- Count occurrence of each word
- Sort in decreasing order
- Display



Type synonyms

- Get words from text:

```
words :: [Char] -> [[Char]]
```



```
type Text = [Char]  
type Word = [Char]
```

```
words :: Text -> [Word]
```

Higher order functions

- Should “The” and “the” be counted as same or different?
- Solution: Convert each letter to lowercase!
- Can you identify a function just by looking at its type signature?



What? `:: (a -> b) -> [a] -> [b]`

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

`map toLower :: Text -> Text`

Currying again!



Sort out all your problems

- Count occurrence of each word
- Sort first?

```
sortWords ["to", "be", "or", "not", "to", "be"]  
         = ["be", "be", "not", "or", "to", "to"]
```

```
sortWords :: [Word] -> [Word]
```

- Count adjacent runs of each word:

```
countRuns ["be", "be", "not", "or", "to", "to"]  
         = [(2, "be"), (1, "not"), (1, "or"), (2, "to")]
```

```
countRuns :: [Word] -> [(Int, Word)]
```



Told you *sorting* is remarkable

- Sort in decreasing order:

```
sortRuns [(2,"be"), (1,"not"), (1,"or"), (2,"to")]  
        = [(2,"be"), (2,"to"), (1,"not"), (1,"or")]
```

```
sortRuns :: [(Int,Word)] -> [(Int,Word)]
```

- But we wanted to take only first n entries:

```
take 2 [(2,"be"), (2,"to"), (1,"not"), (1,"or")]  
      = [(2,"be"), (2,"to")]
```

```
take :: Int -> [a] -> [a]
```

Reminder: Every function is curried by default in Haskell!



I only want to see the output

```
showRun (2, "be")  
= "be: 2\n"
```

String is a
type synonym
for [Char].

```
showRun :: (Int, Word) -> String
```

- We want to do this for each entry in the list:

```
map showRun :: [(Int, Word)] -> [String]
```

- Finally, concatenate the list of Strings into one:

```
concat :: [[a]] -> [a]
```



Common Words in a Nutshell

```
commonWords :: Int -> Text -> String
commonWords n = concat . map showRun . take n .
                  sortRuns . countRuns . sortWords .
                  words . map toLower
```



- This is how typical Haskell programs look!!
 - Mathematics is definitely something you didn't study unnecessarily in school!
- But none of those functions was defined?
 - Some of them are predefined (concat, words, take, map, toLower).
 - Rest (showRun, sortRuns, countRuns, sortWords) we can define.
 - Does the implementation matter much? :-)

