Introduction to Functional Programming and Haskell

COMP2209 - Programming III

Dr Julian Rathke

These slides use modified content from Graham Hutton’s slides for the module textbook Programming in Haskell
Imperative Programming

• Consider the following problem:
  • Add up the squares of first $n$ natural numbers
    $$\text{sqsum } n = 0^2 + 1^2 + 3^2 + \ldots + n^2$$
  • How would we solve this in Java, say?

```java
public int sumsq(int n){
    private int sum = 0;
    for (int i=0; i <= n; i++){
        sum = sum + i*i ;
    }
    return sum;
}
```
Imperative Programming

• But what is happening in the machine when we evaluate that program?
• The value $n$ is stored in memory, there are values for $sum$ and $i$ also stored in memory
• Eventually, the values stored in memory change so that the final calculated value of $sum$ yields the answer.
• Imperative programs are instructions saying how to manipulate memory

```java
public int sumsq(int n){
    private int sum = 0;
    for (int i=0; i <= n; i++) { sum = sum + i*i ; }
    return sum;
}
```
Functional and Implementational Content

- But consider the problem again:
- The **Functional Content** of this problem is simply to take an input value $n$ and return an output value “the sum of the first $n$ squares”
- The **Implementational Content** instead says how this should be achieved:
  - Create locations in memory, increment one iteratively up to $n$ and store the running total in the other
- Imperative Programs tend to combine both Functional and Implementation Content in their code.
- Can we do this differently?
Functional Programming

• Programs contain two aspects in their specification:
  • High-level behaviour and low-level implementation
  • Humans tend to be better at the former, but not the latter

• Key Idea of Functional Programming
  • Concentrate on the high-level “functional” behaviour when writing programs
  • Leave the memory management and implementation details to the compiler
Summing squares functionally

• Let’s look at a solution to the problem of summing the first $n$ squares in the functional programming language Haskell:

```
sumsq n = sum [ x^2 | x <- [0..n] ]
```

• Dude, where’s my program?
• It’s right there, sum up the squares of the the numbers 0 to $n$.
• The beauty of functional programming is that the program is often very close to the input/output specification.
• We have defined a function called `sumsq` we can “run” the program by simply applying the function to a value, e.g.
  • `sumsq 4` returns the value 30
What is Functional Programming?

• There is no canonical definition but we can think of it as a programming style in which the basic method of computation is the application of functions to arguments.

• A functional programming language is a programming language that supports and encourages the functional programming style
  • Examples include: Lisp, Standard ML, OCaml, Haskell

• Of course, one can do functional programming in many other, traditionally imperative, languages also.
  • We’ll look at functional programming in Java and JavaScript later in the module.
  • For now we’ll focus on Haskell.
Features of Haskell

• Concise Programs - few keywords, support for scoping by indentation
• Powerful Type System - types are inferred by the compiler where possible
• List Comprehensions - construct lists by selecting and filtering
• Recursive Functions - efficiently implemented, tail recursive
• Higher-Order Functions - powerful abstraction mechanism to reuse code
• Effectful Functions - allows for side effects such as I/O
• Generic Functions - polymorphism for reuse of code
• Lazy Evaluation - avoids unnecessary computation, infinite data structures
• Equational Reasoning - pure functions have strong correctness properties
Historical Background 1930s

• This is Alonzo Church.
• In 1936 he introduced a formal system to demonstrate the undecidability of first-order logic.
• This formal system is called the lambda calculus.
• Today it forms the basis of all functional programming languages.
Historical Background 1950s

- This is John McCarthy.
- In 1958 he introduced the first functional programming language called Lisp.
- It isn’t a language of pure functions as it still allows assignment of variables (memory).
- It is short for LISp Processing.
- Lists are still prevalent in most functional languages.
- Amazingly, LISP and its variants are still in use today.
Historical Background 1960s

- This is Peter Landin.
- In 1966 he introduced a language called ISWIM (If you see what I mean)
- This language rethinks LISP - recognises the importance of pure functions and variables are removed
- The `where` clause is introduced - which makes its way in to Haskell

See: Landin - The Next 700 Programming Languages
Historical Background 1970s

• This is John Backus
• In 1977 he introduced a language called FP
• This language treats functions as *first-class* entities.
• It promotes using *higher-order* functions and composition. These are key concepts in functional programming.
This is Robin Milner

In 1978 he published a paper on type inference and polymorphic types

This featured in the language ML (Meta Language)

This could be considered the first modern functional programming language
Historical Background 1980s

• This is David Turner
• He is a Lazy guy
• He developed a functional programming language using a lazy evaluation semantics called Miranda
• This allowed for more efficient computation (in places) and to work with infinite data structures
Welcome to the world

- A committee was formed in 1987 to develop a lazy language - Miranda was proprietary software!

- Haskell 1.0 was released in 1990 -

- The next major version was Haskell 98 - a stable, minimal version of the language defined in a 2003 report

- From 2010 an updated version of Haskell was released along with Haskell Platform - libraries
Another example: Quicksort

- Take a list of numbers \( xs \) say, pick some value \( x \) in this list
  - Form \( ls \) as the list of numbers in the list lower than \( x \)
  - Form \( rs \) as the list of numbers in the list greater than \( x \)
- Now recursively sort \( ls \) and \( rs \) and then form the sorted list of \( ls \) followed by \( x \) followed by \( rs \)
- Look how we code this in Haskell?

```haskell
quicksort [] = []
quicksort (x : xs) = quicksort ls ++ [x] ++ quicksort rs
  where ls = [ a | a <- xs, a <= x ]
       rs = [ a | a <- xs, a > x ]
```

Nice!
YOUR QUESTIONS

Next Lecture:
Getting started with Haskell