Haskell Coding and Testing and the COMP2209 Coursework

Abridged from notes by Andy Gravell

November 2019

COMP2209 Programming III

Objectives

- To cover a range of Haskell coding techniques
- To illustrate Haskell test automation
- To advise on coding and debugging practices
- Answer questions on the assignment

Example Problem

- Imagine you are asked to implement factorial
- You are given the required signature

```
fact :: Int -> Int
```

- A number of different solutions are possible
  - using common functional coding practices

Direct Definition

- You can use the library function, product

```
fact1 :: Int -> Int
fact1 n = product [1..n]
```

- This can be tested at the command line

```
*Main> fact1 4
24
```

Simple Recursive Solution

- It is nearly as easy to use recursion

```
fact2 :: Int -> Int
fact2 0 = 1
fact2 n = n * (fact2 (n-1))
```

Curried Solution

- You can curry the direct solution
  - composing product with another library function

```
fact3 :: Int -> Int
fact3 = product . enumFromTo 1
```

Higher Order Function

- Or use a standard higher order function

```
fact4 :: Int -> Int
fact4 n = foldl (*) 1 [1..n]
```

Infinite Stream

- At school, factorial was an infinite sequence

```
factorials :: [Int]
factorials = 1:(zipWith (*) [1..] factorials)
```

- Can now define the factorial function directly

```
fact5 :: Int -> Int
fact5 n = factorials !! n
```

Optimised Version

- The recursive solution (#2) can be optimised
  - an accumulating parameter allows tail recursion

```
fact6 :: Int -> Int
fact6 n = fact6' n 0 1

fact6' :: Int -> Int -> Int -> Int
fact6' n m r =
   if m < n then
      let m' = m+1
      in fact6' n m' (m' * r)
   else r
```

Avoid Premature Optimisation

- The factorial function grows exponentially
- After the first 20 results, it overflows

```
*Main> fact1 21
-1195114496
```

- So there is no point optimising this function
  - always wait until you see performance issues

Horizontal & Vertical Coding Styles

- Some solutions are “one-liners” (horizontal)
- The sixth solution has eight lines (vertical)
  - and includes a helper function
  - and is closest to imperative / procedural code
- In the assignment, use whatever style you like
  - make sure your code must be easy to read
  - if part of it is hard to understand, add a comment
  - you will be awarded marks for its clarity
  - with extra marks for concise & elegant solutions

Incremental Development

- You will need to write many functions
  - each one should be quite small, perhaps < 10 lines
- It is sensible to test each one as you write it
- Capture these tests via test automation
  - could be some one-line function invocations
  - Haskell makes this easy as there is no implicit
  - pure functions don’t need complex test set up
- Automation means you can re-run tests easily
  - catch any “regression” (code that stops working)

Test Automation

- Haskell makes it easy to automate testing
  - define some testing constants and functions
  - apply these, for example at the command line
- There are also Haskell testing frameworks
  - these may be overkill for simple coding exercises
  - but are worth investigating for larger programs
- See eg: QuickCheck, HUnit, tasty, Hspec, …

  https://begriffs.com/posts/2017-01-14-design-use-quickcheck.html

  https://wiki.haskell.org/HUnit_1.0_User%27s_Guide

Test Constants

- You can include test values and functions

```
-- some factorial values
factorial7 = 1*2*3*4*5*6*7
factorial8 = 8*factorial7
factorial9 = 9*factorial8

-- this is not a factorial function
constant1 :: Int -> Int
constant1 n = 1
```

Function for Test Automation

- A function that tests a number of factorials

```
test1 :: (Int -> Int) -> Bool
test1 f =
   f 0 == 1 && f 1 == 1 && f 2 == 2 && f 3 == 6 &&
   f 4 == 24 && f 5 == 120 && f 6 == 720 &&
   f 7 == factorial7 && f 8 == factorial8 &&
   f 9 == factorial9

*Main> test1 fact6
True
```

More Sophisticated Test Automation

- Test functions can use any Haskell technique
  - here, a library function and a lambda function

```
test2upTo :: (Int -> Int) -> (Int -> Int) -> Int -> Bool
test2upTo f g n = all (\m -> f m == g m) [0..n]
```

```
*Main> test2upTo fact3 fact4 20
True
*Main> test2upTo fact5 constant1 20
False
```

Simpler Test Automation

- Simpler test functions will catch many errors
  - here, a direct assertion compares two applications

```
test3 :: (Int -> Int) -> Bool
test3 f = f 10 == 10 * f 9
```

```
*Main> test3 fact4
True
*Main> test3 constant1
False
```

Debugging Haskell

- You can test simple functions at the command line, or using simple test functions as above
- It is good practice to structure solutions this way
  - write short and simple functions with clear names
- If a function is too long to explain or understand it will probably be difficult to debug
  - so keep your code clean
  - break up long functions into simpler ones
  - it is also easier to interpret GHC parsing & type errors

Debugging Haskell (continued)

- You can use interactive debugging in GHCi
- See :help for list of commands available

```
*Main> :break fact6'
*Main> fact6 10
Stopped in Main.fact6', factorialFunctions.hs:(34,5)-(37,10)
    _result :: Int = _
    m :: Int = 0
    n :: Int = 10
    r :: Int = 1
```

Debugging Haskell (continued)

- There are also more sophisticated techniques
  - Debug.Trace.trace
  - Hood / Hugs.Observe
  - tracing tools such as Hat, Hoed
- See https://wiki.haskell.org/Debugging

Summary

- There can be many solutions to a problem
  - even working within the functional style
- Code using short, simple and clear functions
- Write simple test constants and functions
  - you can run these via your main program
  - you can also run tests at the command line
- Many debugging techniques and tools exist
  - it is still better to keep your code simple
  - that way you should spend less time debugging

About the Assignment

- Six programming challenges
  - like the lab exercises, but more thought & coding
  - independent so you can skip one and move on
  - but covering related topics
  - each challenge is worth 5 marks
    - based on the published and unseen test cases
- Five marks also for your coding style
- Five marks for your development & testing
  - a report plus your own Haskell test cases