COMP4075/G54RFP
Real-world Functional Programming:
Coursework Part I

Autumn, Academic Year 2018/19

Henrik Nilsson
School of Computer Science
University of Nottingham

October 19, 2018

1 Introduction

The assessed coursework for the module COMP4075/G54RFP (G54RFP from now on) consists of three parts. The first two, Part I and Part II, are essentially programming exercises directly related to the content covered in the lectures. Part III is a mini programming project on a topic of your own choice related to real-world functional programming theme of the module, interpreted broadly. The weights of the three parts are as follows:

- Part I: 10 %
- Part II: 15 %
- Part III: 25 %

Overall, the coursework is thus worth 50 % of the G54RFP mark.

The G54RFP coursework is to be carried out *individually*. You are welcome to discuss the coursework with friends, in the G54CMP Moodle forum, with the module team, etc., but, in the end, you must solve the problems on your own and demonstrate that you have done so by being able to explain your solutions as well as their wider context.

2 Submission

For information about deadlines, see the module web page. There you can also find links to other

For Part I of the coursework, the following has to be submitted by the deadline:

- A brief written report as specified below.
- The source code of all solutions.

The submission is part physical, part electronic:

- *Physical:* hard copy of the report to the Student Support Centre
- *Electronic:*
  - Electronic copy of the report (PDF). The file should be called `psyxyz-report-partI.pdf`, where `psyxyz` should be replaced by your University of Nottingham user ID.
  - Archive of the source code hierarchy (gzipped TAR, or zip). The archive should be called `psyxyz-src-partI.tgz` or `psyxyz-src-partI.zip`, where `psyxyz` again should be replaced by your University of Nottingham user ID, and it should contain a single top-level directory containing all the other files.

The written report should be structured by task. For each task:

- *Brief* comments about the key idea of the solution, how it works, and any subtle aspects; a *few* sentences to a couple of paragraphs would usually suffice.
- Answers to any theoretical questions.
- The code you wrote or added, with enough context to make an incomplete definition easy to understand. Thus, in cases where you have extended given code, you do not need to include what was given, except small excerpts to provide context if necessary. Indeed, if the give code is lengthy, you are encouraged to keep what you include in t eh report to a minimum.
- Anything extra that the task specifically asks for.

To exemplify the point about added and modified code, if you:

- have added a new function, then include the complete function definition, including the type signature;
- have extended a lengthy function with a few cases, then include the new cases along with immediately surrounding cases to the extent needed to make it clear where the extension was made;
- have added a constructor to a datatype, include the definition and state the name of the extended type explicitly.

3 Assessment and Feedback

Both Part I and Part II are structured by tasks, each carrying a weight: a maximal mark between 0 and 100 such that the weights of all tasks add up to 100. Each *individual* task is assessed on two aspects:

- Correctness:
  - 2 (Good): Solution entirely correct according to the specification, except possibly in some very minor way.
  - 1 (Pass): Solution mostly correct, but fails to entirely meet the specification; AND/OR minor omissions.
  - 0 (Fail): Solution mostly incorrect; AND/OR major omissions.
- Style:
  - 2 (Good): Solution elegant and simple, and thus easy to understand; code is well-written, well-formatted, tidy, good names.
  - 1 (Pass): Solution unnecessarily convoluted; AND/OR coding style has major flaws.
  - 0 (Fail): Solution is incomprehensible; AND/OR coding style is unacceptably poor.

The mark for each task is computed as follows:

$$mark = weight \times \frac{correctness + style}{4}$$

After marking, you will, as feedback, get your written report back with each task assessed according to the scheme above.

4 Getting Started

In the following, it is assumed that you are going to use the Haskell system GHC on the School's Linux/Unix servers or the Linux Teaching Desktop. However, GHC is also available on the School's Windows machines, and for the most part things work the same. Unfortunately, the way additional software is installed on the Windows machine seems to be a bit unstable year on year, so the instructions for Windows in the following could be a bit out of date. But at least it should be possible to launch an interactive Haskell session from the start menu (look for GHCi or WinGHCi) and this can at least be used for some of the tasks. Just be aware that building the compiler relies on auxiliary tools such as the parser generator Happy. If you are using the supplied make files, building the compiler is handled automatically for you and you do not need to worry too much about what tools run in which order. But if you exclusively want to use the interactive environment, you will have to run some of these auxiliary tools manually.

If you do work from within Windows, please read section 4.3 first, as there are some caveats that tend to cause a lot of unnecessary confusion.

It may be possible to use other Haskell implementations, and you could certainly use a different platform, such as Mac OS X if you prefer. But then you need to fetch and install those systems yourself, and you cannot expect the module TAs to provide much if any technical assistance if you run into trouble with your installation. The site `www.haskell.org` is your starting point for most things you might want to know about Haskell, and for downloading Haskell implementations, related tools, and documentation.

4.1 Notes for Working on the Linux Servers

The following assumes that you use one of the School's Windows machines, e.g. in the main lab A32, effectively as a terminal. Log on to your Linux server using the SSH Secure Shell Client or PuTTY. Both are available via the Windows Start Menu: All Programs/UoN Applications/Services and Facilities. At the time of writing, the servers are *avon* for 1st year students, *bann* for 2nd year students and *clyde* for 3rd and 4th year students. To connect, you may have to give the full name of the servers; e.g., *clyde.cs.nott.ac.uk*.

Start the interactive GHC environment by issuing the command `ghci` at the command line prompt:

```
clyde$ ghci
```

Some information about GHCi gets printed, and you'll then get a new prompt:

```
Prelude>
```

From here, you can enter and evaluate Haskell expressions, load Haskell code from files, etc.

You can also edit code on the servers using text editors like Emacs (command `emacs` or `emacs -nw` if you want to use Emacs in text mode) or Vi (command `vi`). Using a terminal multiplexer like Screen (command `screen`; do `man screen` for info) you can start a number of interactive sessions (e.g. GHCi, Emacs, shell) and quickly and easily switch between them, all within one window. Alternatively, you can start a number of SSH sessions in separate windows by invoking the SSH client multiple times.

4.2 Notes for Working on the Linux Teaching Desktop

The Linux Teaching Desktop software installation mirrors the one on the Linux servers, so everything will work the same. Which one to pick mainly depends on if you prefer to work in a Linux desktop environment, or if you prefer to work through a terminal window. If you want to install the VMWare Horizon Client on your own machine to access the Linux Teaching Desktop (the client is available for a wide variety of platforms), head to `https://www.nottingham.cloud/`.

4.3 Notes for Working on the Windows Machines

The Haskell Platform, which includes GHCi, has been installed on the Windows machines in the lab. Just select GHCi from the start menu (under All Programs/UoN Applications/_School of Computer Science/Haskell Platform).

Note that you can navigate around the directory structure using the `:cd` command. For example, to get to the H drive:

```
:cd H:
```

Also, you can set GHCi (if it isn't already) as the default program associated with `.hs` files, so you can load them into GHCi just by clicking on them in a file browser window.

Alternatively, you can use WinGHCi. It allows you do do simple things like loading, editing, and running code through GUI shortcuts. However, the associated editor is Notepad, and as Notepad does not understand Unix line-ending conventions, you may need to work around that one way or another in certain cases; see section 4.3.1.

You can edit Haskell files on the Windows machines using editors like Emacs, XEmacs, or Notepad++ (if installed). They all adapt automatically to different line-ending conventions, but Notepad++ may need some configuration regarding the width of tab stops; see section 4.3.2.

4.3.1 Unix and Windows Line-Ending Conventions

As you may be aware, Unix (and hence also Linux, Solaris, Mac OS, etc.) and Windows use different line-ending conventions for text files. Consequently, you could encounter problems if you switch between systems. In particular, the source code for the G53CMP coursework was created under Linux, and thus uses its line-ending convention. To get around this problem, you can either use a text editor that adapts to the convention used, or you can use programs such as `unix2dos` and `dos2unix` to convert text files from Unix to Windows and vice-versa. You can run these programs (under Linux) by supplying them with the names of one or more files to convert (old files will be overwritten); for example:

```
unix2dos MyFile1.hs MyFile2.hs MyFile3.hs
```

Alternatively, you can specify input-output file pairs; for example:

```
unix2dos -n MyFile-Unix.hs MyFile-Windows.hs
```

In more detail, the Unix convention is to use a single character LF (for "Line Feed", ASCII/UNICODE character 10). The Windows convention is to use a character CR (for "Carriage Return", ASCII/UNICODE character 13) followed by LF. An additional complication is that some languages (e.g. C and Haskell) have some provisions for hiding such platform-dependent differences. For example, the character escape sequence `\n` stands for an abstract newline character that signifies a line ending. *Internally*, this may be (and typically will be, but is not guaranteed to be) mapped to the LF character. However, for input/output purposes (in text mode), this character is mapped to and from the appropriate *external, platform specific* line-ending convention, such as LF on Unix-like platforms and CR+LF on Windows platforms. Other languages (e.g Java) takes a different approach and simply define `\n` to be LF and `\r` to be CR. For more information on these issues, see Wikipedia:

`http://en.wikipedia.org/wiki/Newline`.

Note that the HMTC scanner has been written to work with text files using both Unix and Windows line-ending conventions (to the extent visible: see above regarding platform-specific mapping between internal and external representations). Thus, when you are testing the (extended versions of)

the HMTC compiler on some MiniTriangle source code, it does not matter whether this code was written using Unix or Windows line-ending conventions.

4.3.2 Haskell Layout and the Width of Tab Stops

Another issue concerns assumptions about the width of tab stops, although this is more of a tool issue (in particular, text editors, like Emacs or Notepad++) than an operating system issue. If you are using a Windows-specific editor like Notepad++, it is important that you read the following.

Parsing of Haskell programs take layout (indentation) into account (unless the structure is made explicit using curly braces and semicolons). If tab characters are used in a Haskell file, it becomes a critical question just how wide (in spaces) a tab *stop* is supposed to be, as the presence of a tab character means that the horizontal position of the next character should be aligned with the next tab stop. The Haskell language standard has a precise definition (to ensure that Haskell programs always are interpreted the same way): a tab stop is 8 spaces wide. This is also the default in many text editors, like Emacs.

However, for example Notepad++, which is a popular text editor among Windows users, has a different default: it opts for tab stops being 4 spaces wide. To avoid unnecessary grief caused by this (such as seemingly inexplicable parse errors), it is recommended that you, when editing Haskell source using Notepad++, go to Settings, Preferences, Tab Settings and change the width of tab stops to 8, and that you also tick the box "treat tabs like spaces".

If using Notepad, at least from within WinGHCi, the width of a tab stop seems to default to 8, which is appropriate for Haskell, but as noted above, it seems Notepad cannot handle Unix line-ending conventions, so you might need to convert files from Unix to Windows conventions manually.

5 Tasks

Task I.1 (Weight 20 %)

A problem, due to the mathematician W. R. Hamming, is to write a program that produces an infinite list of numbers with the following properties:

i The list is in ascending order, without duplicates.

ii The list begins with the number 1.

iii If the list contains the number x, then it also contains the numbers $2x$, $3x$, and $5x$.

iv The list contains no other numbers.

Solve the following two problems:

1. Given the following function to merge two ordered lists:

```
merge xxs@(x:xs) yys@(y:ys) | x == y = x : merge xs ys
                            | x < y  = x : merge xs yys
                            | x > y  = y : merge xxs ys
```

   define the infinite list `hamming` of Hamming numbers. (You may find using the function `map` or list comprehensions helpful.)

2. Draw the four cyclic graphs that represent `hamming` after the first 1, 2, 3, and 4 elements have been printed.

Task I.2 (Weight 20 %)

1. Extend the spreadsheet evaluator discussed in one of the lectures with two new forms of expressions to compute the sum and average respectively of a range of cells:

```
data Exp = ...
         | Sum CellRef CellRef
         | Avg CellRef CellRef
```

   The code for the basic evaluator is given in the file `Sheet.hs`. You may find the function `range` and list comprehensions useful.

2. As discussed, this evaluator has a weakness. Explain the problem and suggest a way to fix it. You don't have to write any code, but your answer should clearly explain the key idea.

Task I.3 (Weight 30 %)

Write a function `drop :: Int -> RList a -> RList a` that deletes the first n elements for a *skew* binary random-access list. Your function should run in $O(logn)$ time.

Explain how your implementation works and in particular why it has the desired time complexity. Be sure to test your solution thoroughly and provide some evidence of testing with your answer. The code for skew binary random-access lists is given in the file `SBRAL.hs`.

Task I.4 (Weight 30 %)

Interval arithmetic, as the name suggests, is arithmetic defined on numerical intervals. For example, it can be used to compute error bounds. Say we know $x \in [l_x, u_x]$, and $y \in [l_y, u_y]$, then

$$x + y \in [l_x + l_y, u_x + u_y] \quad \text{and} \quad x - y \in [l_x - u_y, u_x - l_y]$$

Let us represent an interval as follows:

```
data Ivl = Ivl Double Double deriving Show
```

Make `Ivl` an instance of the type classes `Num` and `Fractional` (Methods `(+)`, `(-)`, `(*)`, `abs`, `signum`, `fromInteger` for `Num`; methods `(/)`, `(recip)`, `(fromRational)` for `Fractional`). You should enforce the invariant that for any value `Ivl` l u, $l \leq u$. Use `error` to give suitable error messages when partial operations are undefined.

The above instances will make it possible to use overloaded numerical literals to construct intervals containing only that specific number. E.g. `1` denotes `Ivl 1.0 1.0` when used at type `Ivl`. In addition., define an operator

```
(+/-) :: Double -> Double -> Ivl
```

for constructing symmetric intervals around a specific number. E.g. `1 +/- 0.5` denotes `Ivl 0.5 1.5`.

Test your solution thoroughly and provide some evidence of testing with your answer.