COMP4075
Real-world Functional Programming: Coursework Part II And Exam

Autumn, Academic Year 2020/21

Henrik Nilsson
School of Computer Science
University of Nottingham

December 5, 2020

1 Introduction

The module COMP4075 was set to be assessed by 100 % coursework the academic year 2020/21: a change from previous academic years, where the assessment was by 50 % coursework and 50 % exam, to adapt the module delivery to the circumstances brought about by the Covid pandemic. However, due to a clerical error, the official syllabus for COMP4075 was never updated, meaning that the module still has to be examined by way of coursework and exam. As this problem surfaced well after teaching had started, it has been agreed that the formal assessment requirements will be met by "converting" part of the old Part II Coursework into an on-line, take-home "exam" so as to minimize the impact of this error on the students taking the module.

Concretely, Task 1 of the old coursework Part II will be the new coursework Part II, to be submitted by the original deadline, while tasks 2 and 3 will be the "exam", to be submitted by an examination date yet to be decided in January 2021, but it will be at the earliest 11 January. The upshot of this is that you will have to make two submissions for what used to be Part II, that the weights of the tasks have been changed slightly, but that you have substantially more time to complete the work for the main part. Everything else remains unchanged: in particular what needs to be done and what ultimately needs to be submitted.

The new weights for the different assessment components, as a percentage of the overall module mark are as follows:

- Coursework Part I: 25 %
- Coursework Part II = Task II.1: 25 %
- Exam = Task II.2 & Task II.3; 50 %, with the individual components weighted:
  - Task II.2: 40 %
  - Task II:3: 10 %

The COMP4075 coursework and exam is to be carried out *individually*. You are welcome to discuss the coursework with friends, in the COMP4075 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 Practicalities

For basic practical information on how to work with Haskell on the School's computers, and issues specific to various platforms such as Windows and Linux, see Part I of the coursework.

However, for some of the tasks here you may have to install additional packages using Cabal. To get started, ensure the list of available packages is up to date:

```
cabal update
```

You can list what packages are available; e.g.

```
cabal list
cabal list --installed
```

To install a package, do:

```
cabal install <package-name>
```

However, installing large packages, at least on the School's servers, sometimes fails with errors "failed to create OS thread" (which can be hard to see as this then causes lots of follow-on errors obscuring the initial cause). The problem appears to be that certain resource limits are configured in a quite conservative way on the School's servers. To avoid this problem, limit the

number of concurrent processes that Cabal spawns. The most conservative is to set the limit to 1, but this means no parallelism and thus possibly lengthy installation times:

```
cabal install -j1 <package-name>
```

Setting the limit to 3 also appears to work and might be an acceptable compromise.

For general assistance, Cabal provides built-in help:

```
cabal help
cabal help <subcommand>
```

3 Submission and Assessment

For information about deadlines, see the module web page. For Part II of the coursework and the Exam, the following has to be submitted by the relevant deadline:

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

The submission is electronic. For Part II of the coursework:

- Electronic copy of the report (PDF). The file should be called `psyxyz-report-partII.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-partII.tgz` or `psyxyz-src-partII.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 submission requirements for the exam are the same, except that the exact naming conventions may be different: follw the submission instrutions for the exam.

The written reports 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.
- Unless specified otherwise, 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 the report to a minimum.
- Anything extra that the task specifically asks for.

As for part I, solutions will be assessed on correctness and style. See the Part I description for details. However, for tasks with a large weight, fractional correctness and style marks may be used for a more fine-grained assessment.

4 Tasks

Task II.1 (Weight 25 %)

Implement a deadlock-free solution to the Dining Philosophers problem using Software Transactional Memory (STM). See

```
https://en.wikipedia.org/wiki/Dining_philosophers_problem
```

for the problem statement.

Each philosopher should be represented by a thread (`forkIO`), given a name and, to allow us to see what is going on, announce (print to the terminal) when they are hungry, eating, and thinking, stating their name as well in each case (e.g. `Socrates is hungry`). They should eat and think for a period of time decided at random (import `System.Random` and use `threadDelay`).

While the solution to this task will be rather "imperative" in style, it is still possible to deploy neat functional programming techniques and appropriate types to obtain an elegant overall solution, and this will be taken into account is the assessment for style.

In addition to brief explanations of the design and implementation, discuss your STM solution in relation to the two classical solutions outlined in the Wikipedia article (*Resource Hierarchy Solution* and *Arbitrator Solution*). Explain in particular why your solution is free from deadlocks, and any pros and cons of using STM compared to the classical solutions.

In summary, for his task, the written report should include:

- Brief explanations of your design and implementation
- The code you wrote
- Enough sample output to provide evidence that your solution is working
- Discussion of your STM solution in relation to the classical solutions

Task II.2 (Weight 40 %)

Implement a simple "electronic calculator" using the *Threepenny* GUI framework: `https://wiki.haskell.org/Threepenny-gui` (this will be covered in a lecture shortly). It should look and behave in a reasonably standard way: see section 1 of `https://en.wikipedia.org/wiki/Calculator`.

You will likley have to install the *Threepenny* GUI package:

```
cabal install threepenny-gui
```

However, do check the section on using Cabal above, in particular if the installation fails for some reason. Also, if you are running on the School's Linux servers and if you want interact with your application from a browser running on your local machine, you will have to set up a tunnel:

```
ssh -fN -L8023:127.0.0.1:8023 severn
```

At the very least, the calculator should:

- Handle at least 10-digit integers
- Support addition, subtraction, multiplication, and division
- Allow the sign to be changed (+/-)
- Allow the calculator to be reset (C) as well as clearing the last entry (CE)

A functioning and reasonably looking basic calculator will get half the marks.

For full marks, the calculator should additionally:

- Support calculations with decimal fractions (decimal point)
- Have a memory and associated functions for store and recall
- Support standard precedence rules among the arithmetic operations along with parentheses for grouping (e.g. the result of entering $1+3*4 =$ should be 13 and the result of entering $5 + 3 * (5 - 9) =$ should be -7)
- Have a clearly structured implementation making use of appropriate functional programming techniques and types (e.g. a state transition function embodying the logical core, Functional Reactive Programming or other ways to manage events and effecs to avoid the proverbial "call-back soup").

You might find the classic Shunting-yard Algorithm due to Edsger Dijkstra to be helpful for handling precedence:

```
https://en.wikipedia.org/wiki/Shunting-yard_algorithm
```

Another possibility is to simply gather all input as a string and parse it according to a grammar imparting the desired precedence among the operators. This tend to be how more modern calculators, including typical mobile apps, work. The parsing could happen repeatedly as the input is being entered, with an evaluation after each successful parse, or only at the end when the result is demanded.

For this task, the written report itself should include:

- Brief explanation of your design and implementation, including a clear statemenet of which requirements your solution handles
- Enough code to illustrate the key ideas and aspects of the implementation
- Two or three screen dumps showing your calculator in action

Task II.3 (Weight 10 %)

This task concerns developing some QuickCheck tests for the Skew Binary Random Access List (SBRAL) implementation from part I of the coursework, Task I.3. The QuickCheck tests need to meet the requirements set out below, either by following the suggested approach outlined here, or by developing your own set of tests. Indeed, you may already have used QuickCheck in your solution to Task I.3, in which case you can reuse the relevant parts for this task. You can choose to test either your own solution to task I.3, or the one from the model solution (once released).

Requirements:

- Three different QuickCheck properties, each testing an aspect of genuine interest.
- The report should provide a brief explanation of each property.
- At least one of the properties should cover `drop`, and at least one should cover `update`.
- The QuickCheck `label` mechanism should be used to give give an idea of the test coverage.
- The report should include representative output for each of the three properties.

Suggested approach: An easy way to generate test data and to formulate properties is to use lists and to relate the SBRAL operations to corresponding operations on lists. As the operations are polymorphic in the type of the list elements, it is fine to formulate the tests in terms of some concrete element type, such as `Int`, as the polymorphic type makes it clear that the element type cannot impact on the behaviour. To that end:

- Implement two functions to convert lists to and from skew binary random access lists:
  - `toRList :: [a] -> RList a`

- `fromRList :: RList a -> [a]`

- Converting a list to a skew binary random access list and back should of course yield the exact same list; i.e., the function composition of `fromRList` and `toRlist` is the identity function on lists. Write a quickcheck property that captures this. E.g., its signature could be:

  ```
  prop_FromRListToRList :: [Int] -> Property
  ```

  Use the `label` mechanism to show the lengths of the lists on which this property is tested when invoked from e.g. `quickCheck`.

- Formulate a correctness property for SBRAL `drop` in terms of `drop` on lists: given a list of length n, dropping $m \leq n$ elements from the SBRAL version of the list should mirror dropping m elements from the list:

  ```
  prop_Drop :: [Int] -> Property
  ```

  Again, use the `label` mechanism to show the lengths of the lists and the chosen number of elements to drop.

- In a similar way, formulate a correctness property for SBRAL `update` by relating it to an update on lists (expressed in terms of list operations like `drop` and `take`):

  ```
  prop_Update :: Int -> NonEmptyList Int -> Property
  ```

  Note that updating is not defined for an empty list. Thus it makes sense to formulate the property for non-empty lists only. Again, use the `label` mechanism to show the lengths of the lists and the chosen update positions.