ECS713 Individual Project – Concurrency and Parallelism

Liam Radley 200907291

December 24, 2020

Difficulties Faced

The primary issue came in the form of figuring out how best to structure the customer datatype and how to use the list of customers in the system to allow for updates to the balances. Initially, an approach was taken whereby the balance was left immutable for each customer. This meant that, after every update, a new customer had to be constructed to represent it. This proved to be horrendously inefficient. This was compounded by the process used to update the list of customers. After transfers were made, the idea was to create a new list filled with the two customers that were updated by the transaction, but then also add in the customers that were not updated. While naively attempting to preserve the order that they were stored in, this led to constructing a function that pattern-matched against every possible combination of two customers based on their index. In brief, this idea was far from what was desired, and would not have lent itself well to the idea of parallelism. This problem was alleviated by converting the balance to an MVar Rational as opposed to a rational, as it allowed for a lock to be imposed on the customer while the transaction was ongoing and also allowed for the updating of the balance attribute directly. This saved an obscene amount of pattern matching!

This created an issue, however – because MVars do not interact well with being created as an instance of Show due to the inherent IO component to access the contents of the MVar, a new function had to be created to extract the information of the client. This affects the purity of the program by a lot – the majority of code relies on some level of IO action. This is objectively rather saddening, and was not something I was able to rectify.

A final issue faced is that, while the program is in itself parallelised, it is not any faster than running it concurrently. It has been assumed this is due to the overhead of parallelisation and the benefits would become clearer when running a larger number of iterations, but with a some level of confidence I suggest that there could be an issue within the code preventing it from parallelising successfully.

Design Choices

Aside from the choices on structuring the Customer datatype, there was a decision made to use the Bowie-themed "Siggy Chardust" module, more specifically the *Data.Ratio.Rounding* package, to allow for rounding of Rational numbers to 2 decimal places. This was appropriate for the setting of working with currency, so was adopted and used within the randomAmount function when trying to generate a random amount to for the person to transfer.
A key decision made when working on the project was to use, instead of *Control.Parallel*, the *Control.Concurrent.ParallelIO.Global* module within the "parallelIO" package to run the parallel component of the program, as it provided a much more intuitive method of amalgamating the IO actions one desired to parallelise. All that was required was to place the IO actions into a list, then use the *parallel* function within the module and the program would run in parallel.

How to run

Within the *parallel-banking* directory, run *stack build* which will compile the necessary files within the *src* folder. Then, navigate to the *app* folder. In here, running *"stack ghc Main.hs – -O2 -threaded -rtsopts"* will compile Main.hs and will optimise it for threading. After this, run *./Main* and you will see a pair of arguments that can be inserted onto the command line.
Running *"./Main concurrent"* will run the code using only the Control.Concurrent library, whereas running *"./Main parallel +RTS -N -RTS"* will run the program using parallelism. Note that using the -N flag as opposed to specifying the number allows the number of threads necessary to be decided by the computer at runtime. This is advised when running this program with the parallel argument, as running with anything less than 3 threads is not guaranteed to exit at all, instead getting stuck in an infinite loop once the accounts run out of money. The proof of this is provided in the appendix.
Documentation is available on the relative path *./.stack-work/install/5f37e22e/doc/index.html* from the *parallel-banking* directory.

Appendix

Proposition 0.0.0.1. *The program must be run on at least 6 threads when parallelised in order to ensure it exits.*

Proof. Note that the maximum amount of money that can be transferred throughout the program is $50 \times 100 = 5000$. Assume the program runs on five or less threads. Then, assuming the same number of accounts accounts are able to maintain a lock on the threads in question, then the worst case scenario is that the threads will transfer all of their money to these other five accounts. If this happens, then the dormant accounts will not be able to get a lock on the thread, so cannot execute any transactions, and an infinite loop is created. This does not happen after 6 threads are used, as the total balance of all of the active accounts exceeds the maximum amount that can be transferred. $\square$