Weekly Assignments 2

To be submitted: Friday, 4 August 2017

Note that some tasks may deliberately ask you to look at concepts or libraries that we have not yet discussed in detail. But if you are in doubt about the scope of a task, by all means ask.

Please try to write high-quality code at all times! This means in particular that you should add comments to all parts that are not immediately obvious.

W2.1 Packaging

Prepare a Cabal package to contain all your solutions so that it can easily be built using cabal-install or stack.

Note that one package can contain one library (with arbitrarily many modules) and possibly several executables and test suites.

Please include a `README` file in the end explaining clearly where within the package the solutions to the individual subtasks are located.

For task W2.4, include a text file in the package containing the solution.

The project exercise P1 is not to be included in the package, and handled separately.

Please do *NOT* try to upload your package to Hackage.

W2.2 Dining Philosophers

The Dining philosophers problem is a classic problem to demonstrate synchronization issues in concurrent programming and ways to resolve those problems. *illustration By Benjamin D. Esham / Wikimedia Commons, CC BY-SA 3.0*

`n` philosophers sit aroung a table, between each of them lies a fork. We label philosophers and forks by the numbers from 1 to `n` (counterclockwise), such that fork #1 is the fork between philosophers #1 and #2, fork #2 is the fork between philosophers #2 and #3 and so on.

The philosophers are hungry from their discussions and want to eat. In order to eat, a philosopher has to pick up the fork to his left and the fork to his right. Once he has both forks, he eats, then puts the forks down again, then immediately becomes hungry again and again wants to eat.

We model each philosopher as its own concurrent Haskell thread. For each of the subtasks, log some output to the screen, so that you can observe which philosopher is holding which fork and which philosopher is eating. You may need to come up with a way to ensure that the log output is not garbled, even though all philosopher threads try to log concurrently.

Provide one executable for each subtask, where `n` can be provided as a command line argument. Keep the executables as small as possible and implement most code in the library.

Try to share as much code as possible between subtasks!

Subtask 2.2.1

First we want to demonstrate the danger of *deadlock*. Use an `MVar ()` for each fork, where an empty `MVar` means that the fork has been picked up. Implement the philosophers by having them always pick the left fork first, then the right fork.

Observe that very soon, there will be deadlock.

Subtask 2.2.2

One classic strategy for deadlock avoidance is to impose a *global order* on the order in which locks are taken. Change your deadlock-prone program from above in such a way that now, each philosopher first tries to pick up the fork *with the lower number*, then the fork *with the higher number*.

Observe that the deadlock problem is fixed and that the philosophers can all eat.

Subtask 2.2.3

Finally, we want to see how using `STM` avoids the deadlock problem without the need for any global order of locks: Represent forks as `TVar Bool`'s, where value `False` means that the fork is on the table, `True` means it has been picked up. Have the philosophers try to pick up the left fork first, then the right one.

Observe that again, we have no deadlock.

W2.3 Unsafe IO

In this task, we're going to show that `unsafePerformIO :: IO a -> a` is not just a function that can lead to unpredictable results, but it can in fact be really dangerous and lead to crashes such as segmentation faults.

For this, we're going to combine `unsafePerformIO` with mutable references, so you will need the following modules:

```haskell
import Data.IORef
import System.IO.Unsafe (unsafePerformIO)
```

Subtask 2.3.1

Just to practice working with `IORef`s, write a function

```haskell
relabelTree :: Tree a -> IO (Tree (a, Int))
```

where

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

that assigns unique integer labels from left to right to all of the leaves, starting from `0. This time, we do not want to use theStatetype, but simply make use of an`IORef Int‘ to hold the counter.

In this part, you should *not* use `unsafePerformIO`.

Subtask 2.3.2

Use `unsafePerformIO` to define a value of type

```haskell
anything :: IORef a
```

Can you see why the presence of such a value is dangerous?

Subtask 2.3.3

Use `unsafePerformIO` and `anything` to define a function

```haskell
cast :: a -> b
```

that abandons all type safety. Play with `cast` a bit and see what happens if you cast various types into each other. Can you find cases where this actually works? And why? Can you also find cases where this reliably crashes?

W2.4 Equational reasoning

Assume the following definitions:

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)

size :: Tree a -> Int
size (Leaf a)   = 1
size (Node l r) = size l + size r

flatten :: Tree a -> [a]
flatten (Leaf a)   = [a]
flatten (Node l r) = flatten l ++ flatten r

length :: [a] -> Int
length []       = 0
length (x : xs) = 1 + length xs

(++) :: [a] -> [a] -> [a]
[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

Subtask 2.4.1

Prove the following property:

```
forall (xs :: [a]) (ys :: [a]) .
length (xs ++ ys) = length xs + length ys
```

Subtask 2.4.2

Prove the following property:

```
forall (t :: Tree a) .
length (flatten t) = size t
```

W2.5 Transactions

In most cryptocurrencies, transactions map a number of inputs to a number of outputs, roughly as follows:

```
data Transaction =
  Transaction
    { tId      :: Id
    , tInputs  :: [Input]
    , tOutputs :: [Output]
    }
  deriving (Show)

type Id = Int
```

The `Id` is simply a unique identifier for the transaction.

The idea is that all the inputs are completely consumed, and the money contained therein is redistributed to the outputs.

An `Output` is a value indicating an amount of currency, and an address of a recipient. We use `String` to model addresses, and keep values all as integers.

```haskell
data Output =
  Output
    { oValue   :: Int
    , oAddress :: Address
    }
  deriving (Show)

type Address = String
```

An `Input` must actually refer to a previous output, by providing an `Id` of a transaction and a valid index into the list of outputs of that transaction.

```haskell
data Input =
  Input
    { iPrevious :: Id
    , iIndex    :: Index
    }
  deriving (Show, Eq, Ord)

type Index = Int
```

In processing transactions, we keep track of "unspent transaction outputs" (UTXOs) which is a map indicating which outputs can still be used as inputs:

```haskell
type UTXOs = Map Input Output
```

The outputs contained in this map are exactly the unspent outputs. In order to refer to an `Output`, we need its transaction id and its index, therefore the keys in this map are of type `Input`.

Subtask 2.5.1

A transaction is called *valid* if all its inputs refer to unspent transaction outputs, and if the sum of the values of its outputs is smaller than or equal to the sum of the values of its inputs. (In real cryptocurrencies, the amount by which it is smaller is usually considered to be the transaction fee and not lost, but reassigned to the block creator in a special transaction. But in our small example, that money simply disappears.)

In the light of this, implement a function

```haskell
processTransaction :: Transaction -> UTXOs -> Either String UTXOs
```

that checks if a transaction is valid and at the same time updates the UTXOs by removing the ones that are used by the transaction. If the transaction is invalid, an error message should be produced.

Then, write a function

```
processTransactions :: [Transaction] -> UTXOs -> Either String UTXOs
```

that processes many transactions in sequence and aborts if there is an error.

Subtask 2.5.2

Construct a number of small example transactions and an initial state of unspent transaction outputs (that determines the initial money distribution, because we have no way to create money here), and verify that `processTransactions` behaves as intended.

Subtask 2.5.3

For the previous subtask, you will have to write several functions of the type

```
UTXOs -> Either String (a, UTXOs)
```

This looks like a combination of the `State` type with `Either`.

Let's define

```
newtype ErrorState s a = ErrorState { runErrorState :: s -> Either String (a, s) }
```

Define a `Monad` instance (and the implied `Functor` and `Applicative` instances) that combines the ideas of the monad instances for `Either` and `State`, i.e., it aborts as soon as an error occurs (with an error message), and it threads the state through the computation.

Also define the functions:

```
throwError :: String -> ErrorState s a
get :: ErrorState s s
put :: s -> ErrorState s ()
```

in similar ways as we had done for the individual monads.

Subtask 2.5.4

Rewrite `processTransactions` and all the helper functions to use the `ErrorState` type.

P1 A peer-to-peer network

This is the first project exercise which you should work on as a team. It has a preliminary deadline of Tuesday, 8 August. At that day, you either have to demonstrate the solution or at least provide a status report.

Please work in the following teams:

Team 1

- Alexander
- Andreas
- Costas D.

Team 2

- Costas V.
- Stavros
- Thomas
- Vasillis

The goal is to implement a simple peer-to-peer (p2p) discovery protocol.

The protocol is text (and line) based and roughly given by the following datatype of messages:

```
data Message =
    Connect HostName PortNumber
  | GetPeers
  | Status [Peer]
  | Newtx Tx
  | Oldtx Tx Tx
  | Quit
  | Unknown String
  deriving (Read, Show)
```

We will learn about proper parsing and serialization to binary formats soon, but for this task, you can use `read` and `show` for parsing and serialization.

Transactions are just `Int`s here:

```
type Tx = Int
```

In a p2p setting, every process acts as *both* a server and a client. We try to build a network where several processes connect to each other and propagate a bit of shared state, which is the number of the most recent (maximum) transaction.

The `Connect` message is special: a process has to send this message upon connecting to another process, and it must not sent it again after that. The `Connect` message is supposed to be parameterized by the hostname and port number under which the originating process can be contacted.

The `Quit` message, if sent, should cause the target process to end the connection properly.

The `GetPeers` message can be used to request a `Status` response, in which a process lists its known peer processes.

The `Newtx` message can be used to inform another process about a possible new transaction.

The `Oldtx` message is purely informational and states that a particular transaction is already known to the process, and it also includes the most recent transaction.

The `Unknown` message can be used to signal incorrect inputs, and thereby help debugging erroneous programs.

Every process maintains a list of peers, which are other processes it is connected to, by remembering at least their host names and port numbers. There are two ways in which a process can be connected to another process. It may have initiated the connection itself (after having been informed about a possible peer via a message), or it received an incoming connection.

A process can decide how many peers it wants to be connected to. I recommend to implement at least the following strategy: A process will accept and retain any incoming connection. On startup, a process will be given a possible seed node as a command line parameter. It will then try to connect to that node and ask it for its peers. On receiving the peers, it might connect to these and ask them for their peers as well, and continue this process for a few times (making sure not to unnecessarily connect to the same process over and over again). After the process has either no ways of finding new peers anymore, or reached at least knowledge of a given number of possible peers (e.g. 10), it will randomly pick a small number (e.g. 3) out of these and connect to them.

This way, a loosely connected network can be built by connecting more and more nodes and letting them choose what other nodes to connect to.

Processes should also retain the most recent transaction they've seen. Whenever they receive a new transaction, they should propagate it, but with a (configurable) delay. Propagate means they'll just re-issue the `Newtx` message to their peers. If a process receives a transaction that it already knows or is less recent than its current transaction, it should instead reply with an `Oldtx` message, including both the transaction it rejected and the maximum transaction it knows about. A `Newtx 0` message can be used by a new process to figure out what the peers believe the current transaction is.

Processes should log incoming transactions to a file or the terminal with timestamps. Look at the `time` package on how you can access the system time.

Processes should, at random intervals (between 10 seconds and a few minutes), generate new transactions, between `1` and `10` higher than the most recent transaction they currently know. They should log when they create a new transaction and then send it to their peers.

Processes should deal with error situations properly. If other processes disappear, the network might seize functioning, but at the very least processes should gracefully handle such situations and not crash.

Optional features:

- Every process could also have an interactive interface where you can tell it to send particular messages right now.
- You could extend the messages with `Ping` / `Pong` messages that processes send to their peers in certain intervals to ensure they're still alive.
- If processes disconnect and the number of peers of a node gets below a certain level, the node could re-start the discovery process and try to find new peers to connect to.
- You could try to let processes map out the complete network graph as they perceive it by sending subsequent peer messages and export it as a graph. Look at the `DOT` file format of graphviz to export a graph in an easily drawable format. There are also Hackage graphviz packages, but they're probably overkill for this.
- You could try to collect and aggregate the logs of the different nodes and visualize how transactions propagate through the network.
- Think of other interesting features you might add.

Important:

- Keep your code clean and refactor often. Readability and clarity is more important than having as many features as possible. Documentation also helps.
- Everyone must be able to answer general questions about the code. So even though you work together in a team, talk to each other about the code you write and review each other's code, so that you know what they've written.