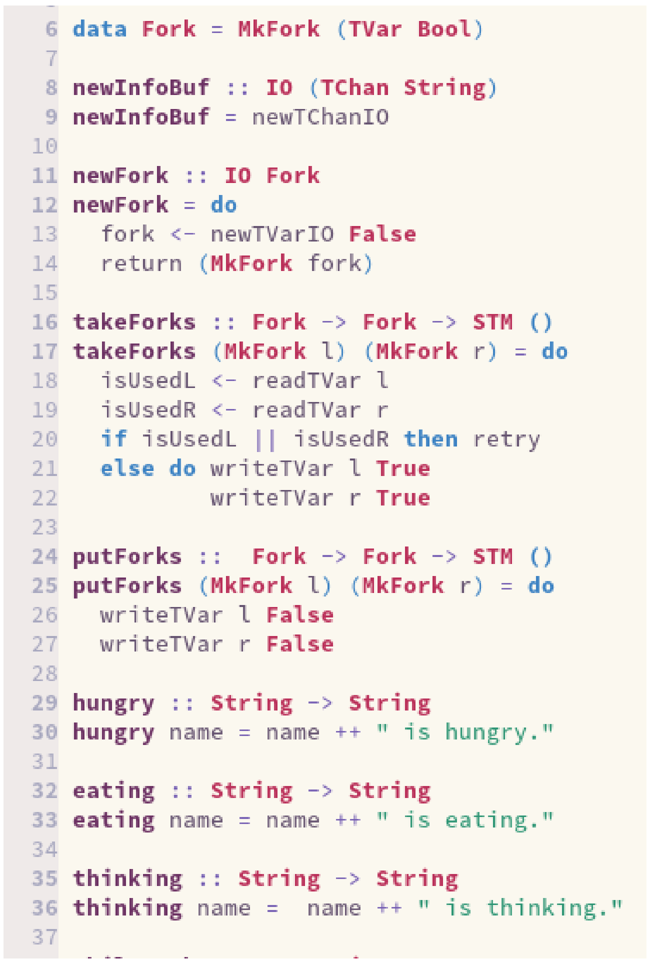
Real-world Functional Programming

Coursework Part II Report

14274056 Junsong Yang (psyjy3)

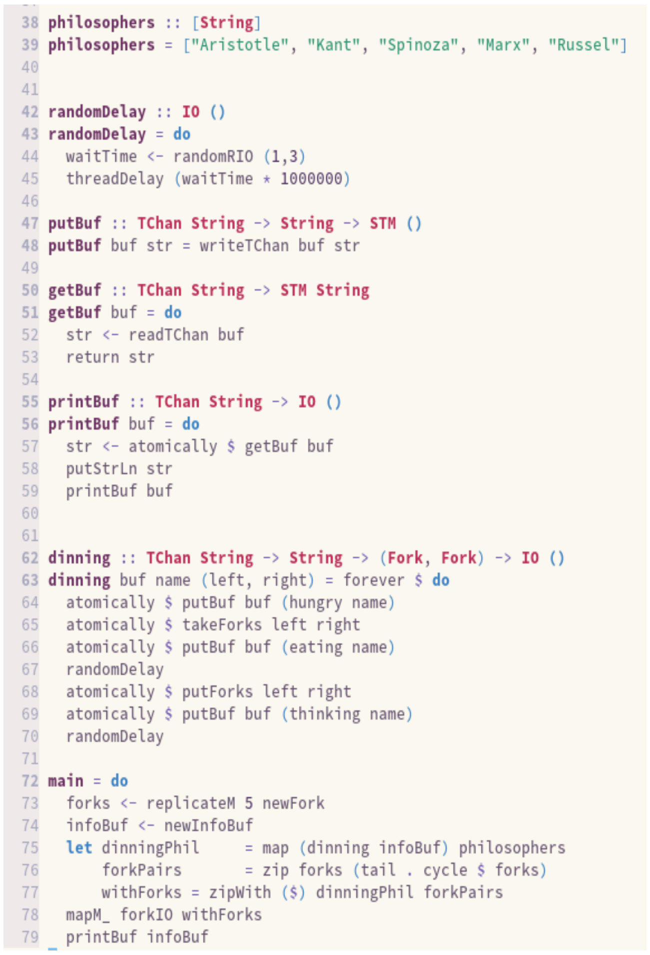
December 10, 2019

1 Task II.1



```haskell
data Fork = MkFork (TVar Bool)

newInfoBuf :: IO (TChan String)
newInfoBuf = newTChanIO

newFork :: IO Fork
newFork = do
  fork <- newTVarIO False
  return (MkFork fork)

takeForks :: Fork -> Fork -> STM ()
takeForks (MkFork l) (MkFork r) = do
  isUsedL <- readTVar l
  isUsedR <- readTVar r
  if isUsedL || isUsedR then retry
  else do writeTVar l True
          writeTVar r True

putForks ::  Fork -> Fork -> STM ()
putForks (MkFork l) (MkFork r) = do
  writeTVar l False
  writeTVar r False

hungry :: String -> String
hungry name = name ++ " is hungry."

eating :: String -> String
eating name = name ++ " is eating."

thinking :: String -> String
thinking name =  name ++ " is thinking."
```

(a) Dinning Phhilosopher Part I



```haskell
philosophers :: [String]
philosophers = ["Aristotle", "Kant", "Spinoza", "Marx", "Russel"]


randomDelay :: IO ()
randomDelay = do
  waitTime <- randomRIO (1,3)
  threadDelay (waitTime * 1000000)

putBuf :: TChan String -> String -> STM ()
putBuf buf str = writeTChan buf str

getBuf :: TChan String -> STM String
getBuf buf = do
  str <- readTChan buf
  return str

printBuf :: TChan String -> IO ()
printBuf buf = do
  str <- atomically $ getBuf buf
  putStrLn str
  printBuf buf


dinning :: TChan String -> String -> (Fork, Fork) -> IO ()
dinning buf name (left, right) = forever $ do
  atomically $ putBuf buf (hungry name)
  atomically $ takeForks left right
  atomically $ putBuf buf (eating name)
  randomDelay
  atomically $ putForks left right
  atomically $ putBuf buf (thinking name)
  randomDelay

main = do
  forks <- replicateM 5 newFork
  infoBuf <- newInfoBuf
  let dinningPhil     = map (dinning infoBuf) philosophers
      forkPairs       = zip forks (tail . cycle $ forks)
      withForks = zipWith ($) dinningPhil forkPairs
  mapM_ forkIO withForks
  printBuf infoBuf
```

(b) Dinning Phhilosopher Part II

Figure 1: Dinning Phhilosopher

Figure 4 contains two figures that show the full implementation of dinning philosopher using STM. Starting with figure 4 (a), the Fork type is defined with a constructor called MkFork which take a TVar with a Bool as a input. If the boolean inside the TVar is false then this fork is available to be used otherwise this fork is already taken. The newInfoBuf function will return a TChan with String wrapped in IO monad to be used later to store logs that indicate the running state of each thread. The newFork function will return a Fork wrapped in IO monad with the boolean value set to false.

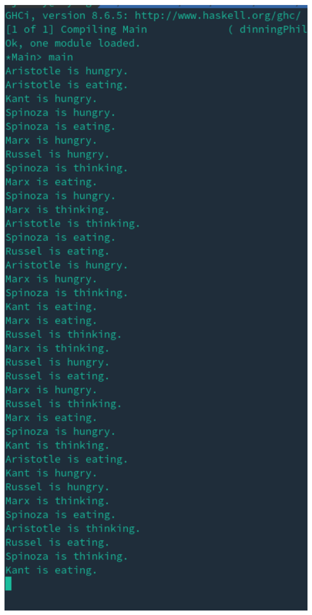
The next two function takeForks and putForks are related to require and release resources. The takeForks function takes two Fork as input. This function will first check if the two Forks are both available. The two Forks can only be used if they are both available as the same time. Otherwise, this function will keep retrying untill both Forks can be required. The putForks function is simple just release the two Forks by setting the boolean to true.

The next three functions hungry, eating, thinking are just dummy function that concatenate the name of philosopher with corresponding information. These information will later be put into the infoBuf(TChan String). The names of philosophers are defined as a list of string as figure 4 (b) shows.

The randomDelay will call threadDelay to delay the running thread randomly from 1 to 3 seconds. The next three functions, putBuf, getBuf printBuf are operations related to the infoBuf(TChan String) that is used to store the logs for running thread. The putBuf function will just store string to the infoBuf(TChan String), the getBuf function will return the string stored in TChan and wrap in STM monad. The printBuf will just print the string stored in TChan.

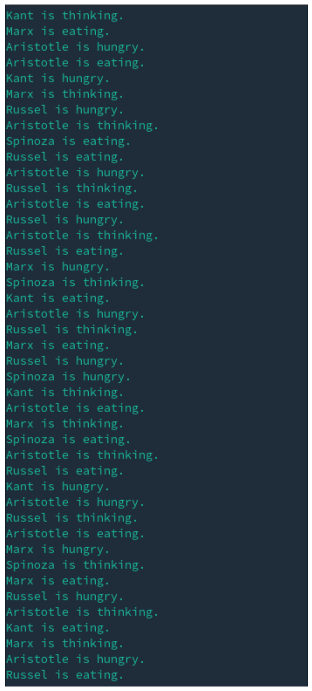
The dinning function is contains the implementation of dinning philosophers. This function takes a TChan String(used to store logs), a string(indicates philosopher's names), and a pair of Fork as input. This function will first store a log in the TChan that indicates the philosopher is hungry. Then, this function will trying to acquire the Fork using takeForks function. If the forks are acquired successfully, another log will be stored in the TChan that suggests the philosopher is eating. Followed by a random delay from 1 to 3 seconds, the forks will be released using putForks function and corresponding log will be put into the TChan. Finally, the function ended with another random delay.

The main function will first initialise 5 Forks using newFork function and a infoBuf using newInfoBuf functions. Then the philosophers' name will be bundled with the dinning function using map. Then, a infinite list of pair of forks will be generated such that each fork in the pair is distinct from the other. Followed by coupling pairs of forks with the dinning function, a list of runnable functions is made. Finally, the main function will run those function by mapping forkIO function to the runnable dinning functions while the printBuf function is called to print the logs of those running thread.



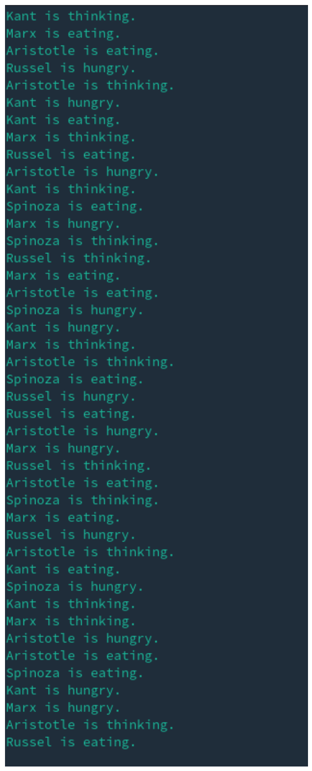
```
GHCi, version 8.6.5: http://www.haskell.org/ghc/
[1 of 1] Compiling Main             ( dinningPhil
Ok, one module loaded.
*Main> main
Aristotle is hungry.
Aristotle is eating.
Kant is hungry.
Spinoza is hungry.
Spinoza is eating.
Marx is hungry.
Russel is hungry.
Spinoza is thinking.
Marx is eating.
Spinoza is hungry.
Marx is thinking.
Aristotle is thinking.
Spinoza is eating.
Russel is eating.
Aristotle is hungry.
Marx is hungry.
Spinoza is thinking.
Kant is eating.
Marx is eating.
Russel is thinking.
Marx is thinking.
Russel is hungry.
Russel is eating.
Marx is hungry.
Russel is thinking.
Marx is eating.
Spinoza is hungry.
Kant is thinking.
Aristotle is eating.
Kant is hungry.
Russel is hungry.
Marx is thinking.
Spinoza is eating.
Aristotle is thinking.
Russel is eating.
Spinoza is thinking.
Kant is eating.
```

(a) running at 10s



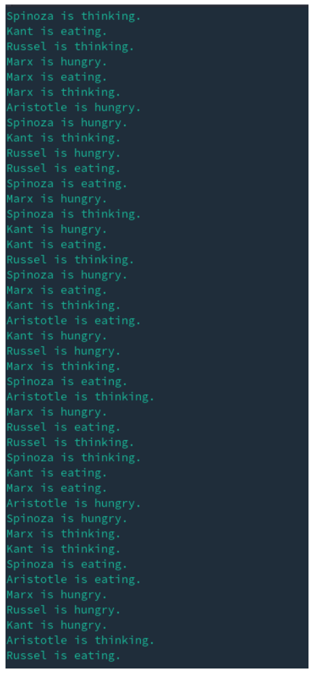
```
Kant is thinking.
Marx is eating.
Aristotle is hungry.
Aristotle is eating.
Kant is hungry.
Marx is thinking.
Russel is hungry.
Aristotle is thinking.
Spinoza is eating.
Russel is eating.
Aristotle is hungry.
Russel is thinking.
Aristotle is eating.
Russel is hungry.
Aristotle is thinking.
Russel is eating.
Marx is hungry.
Spinoza is thinking.
Kant is eating.
Aristotle is hungry.
Russel is thinking.
Marx is eating.
Russel is hungry.
Spinoza is hungry.
Kant is thinking.
Aristotle is eating.
Marx is thinking.
Spinoza is eating.
Aristotle is thinking.
Russel is eating.
Kant is hungry.
Aristotle is hungry.
Russel is thinking.
Aristotle is eating.
Marx is hungry.
Spinoza is thinking.
Marx is eating.
Russel is hungry.
Aristotle is thinking.
Kant is eating.
Marx is thinking.
Aristotle is hungry.
Russel is eating.
```

(b) running at 1m



```
Kant is thinking.
Marx is eating.
Aristotle is eating.
Russel is hungry.
Aristotle is thinking.
Kant is hungry.
Kant is eating.
Marx is thinking.
Russel is eating.
Aristotle is hungry.
Kant is thinking.
Spinoza is eating.
Marx is hungry.
Spinoza is thinking.
Russel is thinking.
Marx is eating.
Aristotle is eating.
Spinoza is hungry.
Kant is hungry.
Marx is thinking.
Aristotle is thinking.
Spinoza is eating.
Russel is hungry.
Russel is eating.
Aristotle is hungry.
Marx is hungry.
Russel is thinking.
Aristotle is eating.
Spinoza is thinking.
Marx is eating.
Russel is hungry.
Aristotle is thinking.
Kant is eating.
Spinoza is hungry.
Kant is thinking.
Marx is thinking.
Aristotle is hungry.
Aristotle is eating.
Spinoza is eating.
Kant is hungry.
Marx is hungry.
Aristotle is thinking.
Russel is eating.
```

(c) running at 2m



```
Spinoza is thinking.
Kant is eating.
Russel is thinking.
Marx is hungry.
Marx is eating.
Marx is thinking.
Aristotle is hungry.
Spinoza is hungry.
Kant is thinking.
Russel is hungry.
Russel is eating.
Spinoza is eating.
Marx is hungry.
Spinoza is thinking.
Kant is hungry.
Kant is eating.
Russel is thinking.
Spinoza is hungry.
Marx is eating.
Kant is thinking.
Aristotle is eating.
Kant is hungry.
Russel is hungry.
Marx is thinking.
Spinoza is eating.
Aristotle is thinking.
Marx is hungry.
Russel is eating.
Russel is thinking.
Spinoza is thinking.
Kant is eating.
Marx is eating.
Aristotle is hungry.
Spinoza is hungry.
Marx is thinking.
Kant is thinking.
Spinoza is eating.
Aristotle is eating.
Marx is hungry.
Russel is hungry.
Kant is hungry.
Aristotle is thinking.
Russel is eating.
```

(b) running at 4m

Figure 2: Dinning Philosopher Running at Defferent Point of Time

Figure 2 contains four figures that shows this implementation running non-stop for four minutes. These sample output of the running program suggests that this implementation is working and without deadlocks. The main reason that this implementation is free of deadlocks is that it does not use locks at all and also this implementation uses STM which allows the threads running without the knowledge of the global environment. Giving two forks, each thread simple keep trying to acquired them at the same time until success. In this way, deadlocks can be avoid as resources will be acquired only until they are available. In contrast, if a thread acquired partially available and wait until other resource is available then deadlocks may easily occur.

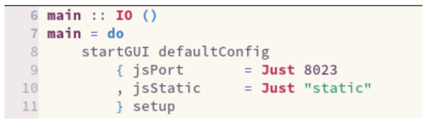
There are other solutions like resource hierarchy solution and arbitrator solution available for this problem. The resource hierarchy solution follows conventions that all forks are acquired in order and only one philosopher can pick up the fork in the highest order. This solution require that all resources are known to all the philosophers. But this is often hard to achieve in real-world use cases which makes this solution impractical. As for the arbitrator solution, resource acquisition is controlled by a arbitrator instead of philosophers. While philosophers can put down forks at any time, the permission of picking up forks is controlled by the arbitrator and the arbitrator will grant the permission to only one philosophers at a time. One obvious issue for this solution is the permission control. All philosophers have to wait for the permission from the arbitrator even if there are forks available. This issue would often cause efficiency problems.

Comparing to those two solutions, STM solution is free of deadlocks and also the code is simple to write and expressive. But the STM solution is suffering from time penalty for committing atomic transactions. Also, one limitation of STM need to be considered is that all the operations that can be performed by STM must be revertible. Despite its disadvantages and limits, STM is justified by its benefits.

2 Task II.2

Before the explanation of this implementation, the capabilities of this calculator will be introduced. The features supported by this calculator are:

- 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)
- Support calculations with decimal fractions (decimal point)
- Support standard precedence rules among the arithmetic operations along with parentheses for grouping
- Have a clearly structured implementation making use of functional reactive programming

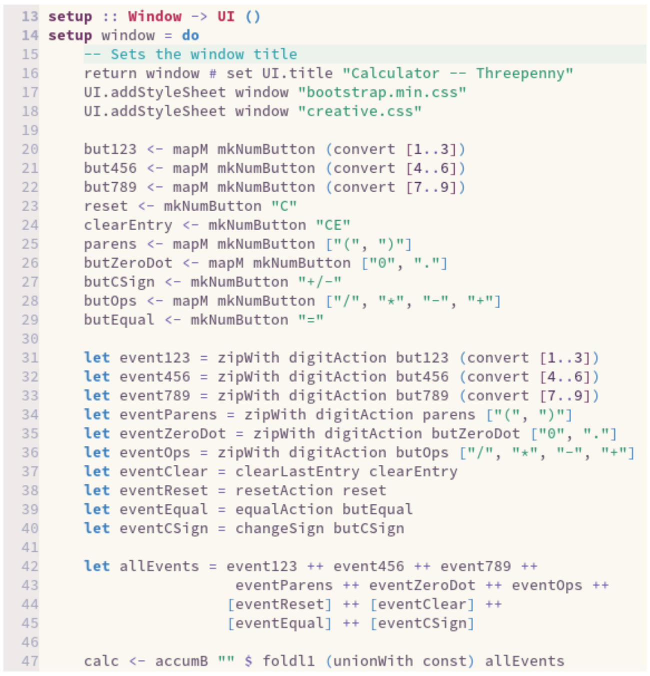


```haskell
main :: IO ()
main = do
    startGUI defaultConfig
        { jsPort       = Just 8023
        , jsStatic     = Just "static"
        } setup
```

Figure 3: Calculator Main Function

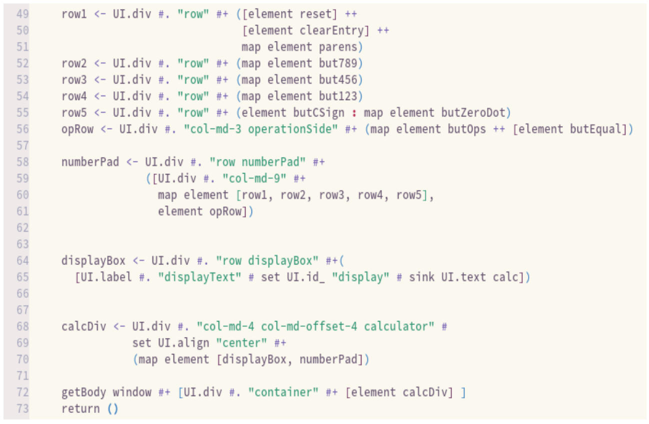
Figure 3 show the main function of the calculator. This function is adopted from the examples provided by the threepenny library.

The setup function is where the logic and ui are defined.



```haskell
setup :: Window -> UI ()
setup window = do
    -- Sets the window title
    return window # set UI.title "Calculator -- Threepenny"
    UI.addStyleSheet window "bootstrap.min.css"
    UI.addStyleSheet window "creative.css"

    but123 <- mapM mkNumButton (convert [1..3])
    but456 <- mapM mkNumButton (convert [4..6])
    but789 <- mapM mkNumButton (convert [7..9])
    reset <- mkNumButton "C"
    clearEntry <- mkNumButton "CE"
    parens <- mapM mkNumButton ["(", ")"]
    butZeroDot <- mapM mkNumButton ["0", "."]
    butCSign <- mkNumButton "+/-"
    butOps <- mapM mkNumButton ["/", "*", "-", "+"]
    butEqual <- mkNumButton "="

    let event123 = zipWith digitAction but123 (convert [1..3])
    let event456 = zipWith digitAction but456 (convert [4..6])
    let event789 = zipWith digitAction but789 (convert [7..9])
    let eventParens = zipWith digitAction parens ["(", ")"]
    let eventZeroDot = zipWith digitAction butZeroDot ["0", "."]
    let eventOps = zipWith digitAction butOps ["/", "*", "-", "+"]
    let eventClear = clearLastEntry clearEntry
    let eventReset = resetAction reset
    let eventEqual = equalAction butEqual
    let eventCSign = changeSign butCSign

    let allEvents = event123 ++ event456 ++ event789 ++
                     eventParens ++ eventZeroDot ++ eventOps ++
                    [eventReset] ++ [eventClear] ++
                    [eventEqual] ++ [eventCSign]

    calc <- accumB "" $ foldl1 (unionWith const) allEvents
```

(a) Part I



```haskell
    row1 <- UI.div #. "row" #+ ([element reset] ++
                                 [element clearEntry] ++
                                 map element parens)
    row2 <- UI.div #. "row" #+ (map element but789)
    row3 <- UI.div #. "row" #+ (map element but456)
    row4 <- UI.div #. "row" #+ (map element but123)
    row5 <- UI.div #. "row" #+ (element butCSign : map element butZeroDot)
    opRow <- UI.div #. "col-md-3 operationSide" #+ (map element butOps ++ [element butEqual])

    numberPad <- UI.div #. "row numberPad" #+
                 ([UI.div #. "col-md-9" #+
                   map element [row1, row2, row3, row4, row5],
                   element opRow])


    displayBox <- UI.div #. "row displayBox" #+(
      [UI.label #. "displayText" # set UI.id_ "display" # sink UI.text calc])


    calcDiv <- UI.div #. "col-md-4 col-md-offset-4 calculator" #
               set UI.align "center" #+
               (map element [displayBox, numberPad])

    getBody window #+ [UI.div #. "container" #+ [element calcDiv] ]
    return ()
```

(b) Part II

Figure 4: Calculator Setup Function

css [0]

Figure 5: newtype Bounded

(a) Recursive Statistics for newtype

(b) Statistics using foldMap for newtype

Figure 6: TaskI.5 3

[0]https://github.com/xxczaki/calculator.js/blob/5d8ec0901e3503a6480d086df0d8e55c39908cdc/css/creative.css