Haskell/Concurrency Braindump

Liam O'Connor
(CSE, NICTA, etc.)

Haskell

- Haskell is a programming language

```
dotp :: [Float] -> [Float] -> Float
dotp l1 l2 = sum (zipProduct l1 l2)
  where
    sum []     = 0
    sum (x:xs) = x + sum xs

    zipProduct (a:as) (b:bs)
     = a * b : zipProduct as bs
    zipProduct _ _ = []
```

Haskell

- Haskell is a *higher order* programming language

```
dotp :: [Float] -> ([Float] -> Float)
dotp l1 l2 = foldr (+) 0 (zipWith (*) l1 l2)
  where
    foldr f z []     = z
    foldr f z (x:xs) = foldr f (f x z) xs

    zipWith f (a:as) (b:bs)
     = f a b : zipWith f as bs
    zipWith f _ _ = []
```

Haskell

- Haskell is a programming language, with amazing libraries for data parallelism:
- On multicore (using REPA):

```
dotp :: Array DIM1 Float -> Array DIM1 Float
     -> IO (Array DIM1 Float)
dotp l1 l2 = R.computeP
           $ R.fold (+) 0 (R.zipWith (*) l1 l2)
```

Haskell

- Haskell is a programming language, with amazing libraries for data parallelism:
- On GPUs (using accelerate):

```
dotp :: Vector Float -> Vector Float
     -> IO (Float)
dotp l1 l2 = A.runAcc
           $ A.fold (+) 0
           $ A.zipWith (*) (use l1) (use l2)
```

Haskell

- For easy task parallelism, Haskell has parallel annotations.
  (you could also use this for data parallelism but the dedicated array libraries perform better)

```
mean :: [Float] -> Float
mean l = sum l / length l
```

Haskell

- For easy task parallelism, Haskell has parallel annotations.
  (you could also use this for data parallelism but the dedicated array libraries perform better)

```
mean :: [Float] -> Float
mean l = let s = sum l
             l = length l
          in s / l
```

Haskell

- For easy task parallelism, Haskell has parallel annotations.
  (you could also use this for data parallelism but the dedicated array libraries perform better)

```
mean :: [Float] -> Float
mean l = let s = sum l
             l = length l
          in s `par` l `pseq` s / l
```

Haskell

- Haskell’s type system keeps this deterministic:

```
example x = let a = (x := 10)
                b = (x := 20)
             in a `par` b `pseq` x
```

Haskell

- Haskell’s type system keeps this deterministic:

side effects!

```
example x = let a = (x := 10)
                b = (x := 20)
             in a `par` b `pseq` x
```

Aside: Purely Functional

- All *functions* are referentially transparent
- *Actions* (computations that can perform side-effects) are represented as a datatype, composed with some sequencing functions.

```
transfer :: Account -> Account -> Int -> IO ()
transfer from to amount = do
  withdraw from amount
  deposit to amount
```

Aside: Purely Functional

- With actions represented differently from functions, functions become a kind of macro language for actions - we can use this to define our own control structures:

```
forever :: IO () -> IO a
forever act = do act; forever act
```

Aside: Aside: Monadic IO

```
(>>=)  :: IO a -> (a -> IO b) -> IO b
return :: a -> IO a
```

```
hEchoLine :: Handle -> IO String
hEchoLine h = do
    s <- hGetLine h
    putStrLn ("I just read that " ++ s)
    return s
```

Aside: Aside: Monadic IO

```
(>>=)  :: IO a -> (a -> IO b) -> IO b
return :: a -> IO a
```

```
hEchoLine :: Handle -> IO String
hEchoLine h = hGetLine h >>=
  \s  -> putStrLn ("I just read that " ++ s) >>=
  \() -> return s
```

Aside: Aside: Monadic IO

```
(>>=)  :: IO a -> (a -> IO b) -> IO b
return :: a -> IO a
```

```
hEchoLine :: Handle -> IO String
hEchoLine h = hGetLine h >>=
  \s  -> putStrLn ("I just read that " ++ s) >>=
  \() -> return s
```

Aside: Aside: Aside: Monads

```
(>>=)  :: m a -> (a -> m b) -> m b
return :: a -> m a
```

Define *Kleisli composition* as:

```
(<.>) :: (b -> m c) -> (a -> m b) -> (a -> m c)
(f <.> g) x = f x >>= g
```

Then Kleisli composition must form a *Category:*

```
f <.> return = f
return <.> f = f
f <.> (g <.> x) = (f <.> g) <.> x
```

Concurrency in Haskell

```
forkIO :: IO () -> IO ThreadId
```

```
forkBomb :: IO () -> IO a
forkBomb = forever (forkIO forkBomb >> return ())
```

Bank account example

- The procedure must operate correctly in a concurrent program, in which many threads may call transfer simultaneously.

```
type Account = MVar Int

transfer :: Account -> Account -> Int -> IO ()
transfer from to amount = do
  withdraw from amount
  deposit to amount

withdraw :: Account -> Int -> IO ()
withdraw account amount = do
  bal <- takeMVar account
  putMVar (bal - amount) account
```

Getting too interesting

- No thread should be able to observe a state in which the money has left one account, but not arrived in the other (or vice versa).

```haskell
type Account = MVar Int

transfer :: Account -> Account -> Int -> IO ()
transfer from to amount = do
  bf <- takeMVar from
  bt <- takeMVar to
  putMVar (bf - amount) from
  putMVar (bt + amount) to
```

Getting too interesting

- No thread should be able to observe a state in which the money has left one account, but not arrived in the other (or vice versa).

```haskell
type Account = MVar Int

transfer :: Account -> Account -> Int -> IO ()
transfer from to amount = do
  bf <- takeMVar from
  bt <- takeMVar to
  putMVar (bf - amount) from
  putMVar (bt + amount) to
```

Deadlock

Global Lock Ordering

```haskell
type Account = (Int, MVar Int)

transfer :: Account -> Account -> Int -> IO ()
transfer (fid, from) (tid, to) amount =
  if fid < tid then do
    bf <- takeMVar from
    bt <- takeMVar to
    putMVar (bf - amount) from
    putMVar (bt + amount) to
  else do
    bt <- takeMVar to
    bf <- takeMVar from
    putMVar (bt + amount) to
    putMVar (bf - amount) from
```

Global Lock Ordering

```haskell
type Account = (Int, MVar Int)

transfer :: Account -> Account -> Int -> IO ()
transfer (fid, from) (tid, to) amount =
  if fid < tid then do
    bf <- takeMVar from
    bt <- takeMVar to
    putMVar (bf - amount) from
    putMVar (bt + amount) to
  else do
    bt <- takeMVar to
    bf <- takeMVar from
    putMVar (bt + amount) to
    putMVar (bf - amount) from
```

"We need it to transfer from a third account, `from2`, if `from` doesn't contain sufficient funds"

Locks are bad, m’kay

- **How you can screw up locks:** Take too few locks, take too many locks, take the wrong locks, take locks in the wrong order, recover inappropriately from errors, forget to wake up threads on condition variables...

  Languages with an emphasis on safety like Haskell are supposed to rule out bugs statically, but none of these bugs can be.

- **Locks are non-modular**

Back to our example

```
transfer :: Account -> Account -> Int -> IO ()
transfer from to amount = do
  withdraw from amount
  deposit to amount


withdraw :: Account -> Int -> IO ()
withdraw from amount =
  modifyIORef (subtract amount) (balance from)

balance :: Account -> IORef Int

modifyIORef ::(a -> a) -> IORef a -> IO ()
modifyIORef f r = readIORef r >>= writeIORef r . f

readIORef  :: IORef a -> a
writeIORef :: IORef a -> a -> IO ()
```

Another monad

IO	STM
`IORef`	`TVar`
readIORef *writeIORef* *modifyIORef*	*readTVar* *writeTVar* *modifyTVar*
atomically :: STM a -> IO a	

Now with STM

```
transfer :: Account -> Account -> Int -> STM ()
transfer from to amount = do
  withdraw from amount
  deposit to amount


withdraw :: Account -> Int -> STM ()
withdraw from amount = modifyTVar (subtract amount) (balance from)

balance :: Account -> TVar Int

modifyTVar ::(a -> a) -> TVar a -> STM ()
readTVar  :: TVar a -> a
writeTVar :: TVar a -> a -> STM ()
```

Guarantees

- Atomicity: the effects of `atomically act` become visible to another thread all at once. This ensures that no other thread can see a state in which money has been deposited in to but not yet withdrawn from from.

- Isolation: during a call `atomically act`, the action `act` is completely unaffected by other threads. It is as if act takes a snapshot of the state of the world when it begins running, and then executes against that snapshot.

  - `atomically (withdraw cashAccount 100)`
  - `atomically (transfer bob jane 100)`
  - `atomically (deposit savings 100)`

Implementation

- Optimistic execution, like in a database.
- When `(atomically act)` is performed:
  - A thread-local transaction log is allocated, initially empty.
  - Then the action `act` is performed, without taking any locks.
  - While performing `act`, each call to `writeTVar` writes the address of the `TVar` and its new value into the log; it does not write to the `TVar` itself. Each call to `readTVar` first searches the log.
  - When the action finishes the implementation first validates the log and, if validation is successful, commits the log (with locks or CAS or what have you).
  - If validation fails, we try the whole transaction again.

Embedding IO inside STM

- Can’t embed IO inside transactions:

```
atomically (do x <- readTVar xv
               y <- readTVar yv
               if x>y then launchMissiles
                      else return () )
```

- Can decide what IO to do inside transactions:

```
act <- atomically (do x <- readTVar xv
                      y <- readTVar yv
                      return $ if x>y then launchMissiles
                                      else return () )
act
```

Blocking

-
```
limitedWithdraw :: Account -> Int -> STM ()
limitedWithdraw acc amount = do
   bal <- readTVar (balance acc)
   if amount > 0 && amount > bal
      then retry
      else writeTVar (balance acc) (bal - amount)
```

```
retry :: STM a
```

Choice

•

```
limitedWithdraw2 :: Account -> Account -> Int -> STM ()
-- (limitedWithdraw2 acc1 acc2 amt) withdraws amt from acc1,
-- if acc1 has enough money, otherwise from acc2.
-- If neither has enough, it retries.
limitedWithdraw2 acc1 acc2 amt
       = limitedWithdraw acc1 amt
`orElse` limitedWithdraw acc2 amt
```

```
orElse :: STM a -> STM a -> STM a
```

The Santa Claus Problem

- **Originally due to Trono:**

  Santa repeatedly sleeps until wakened by either all of his nine reindeer, back from their holidays, or by a group of three of his ten elves. If awakened by the reindeer, he harnesses each of them to his sleigh, delivers toys with them and finally unharnesses them (allowing them to go off on holiday). If awakened by a group of elves, he shows each of the group into his study, consults with them on toy R&D and finally shows them each out (allowing them to go back to work). Santa should give priority to the reindeer in the case that there is both a group of elves and a group of reindeer waiting.

- **Trono gives semaphore-based (partial) solution**

- **Ben-Ari gives solution in Ada**

- **Benton gives solution in C#**

Elves and Reindeers

- Santa has a `Group` for elves and a `Group` for reindeer

```
elf :: Group -> Int -> IO ()
elf group = forever $ do
    (in_gate, out_gate) <- joinGroup group
    passGate in_gate
    meetInStudy
    passGate out_gate

reindeer :: Group -> Int -> IO ()
reindeer group = forever $ do
    (in_gate, out_gate) <- joinGroup group
    passGate in_gate
    deliverPresents
    passGate out_gate
```

Gates

- For a gate created by `newGate n`, all processes will block on `passGate` until someone calls `operateGate`, which allows exactly `n` processes through the gate.

- *`operateGate`* resets the remaining capacity to *`n`* (and thus unblocks every *`passGate`*), and blocks until the remaining capacity is zero again.

```
newGate     :: Int -> STM Gate
passGate    :: Gate -> IO ()
operateGate :: Gate -> IO ()
```

Gates

```
data Gate = MkGate Int (TVar Int)

newGate :: Int -> STM Gate
newGate n = do
    tv <- newTVar 0
    return (MkGate n tv)

passGate :: Gate -> IO ()
passGate (MkGate n tv)
  = atomically (do n_left <- readTVar tv
                   check (n_left > 0)
                   writeTVar tv (n_left-1))

operateGate :: Gate -> IO ()
operateGate (MkGate n tv) = do
    atomically (writeTVar tv n)
    atomically (do n_left <- readTVar tv
                   check (n_left == 0))
```

Group

- A group with initial capacity `n` is created by `newGroup n`
- joinGroup is called by elves and reindeers. It gives access to the `Gates` and decrements the remaining capacity, blocking if it is zero.
- `awaitGroup` is called by Santa. It waits for the group to be full (remaining capacity = 0), then reinitialises the group with new gates. Why?

```
newGroup   :: Int -> IO Group
joinGroup  :: Group -> IO (Gate,Gate)
awaitGroup :: Group -> STM (Gate,Gate)
```

Group

```
data Group = MkGroup Int (TVar (Int, Gate, Gate))

newGroup n = atomically (do g1 <- newGate n; g2 <- newGate n
                            tv <- newTVar (n, g1, g2)
                            return (MkGroup n tv))

joinGroup (MkGroup n tv)
  = atomically (do (n_left, g1, g2) <- readTVar tv
                   check (n_left > 0)
                   writeTVar tv (n_left-1, g1, g2)
                   return (g1,g2))

awaitGroup (MkGroup n tv) = do
    (n_left, g1, g2) <- readTVar tv
    check (n_left == 0)
    new_g1 <- newGate n; new_g2 <- newGate n
    writeTVar tv (n,new_g1,new_g2)
    return (g1,g2)
```

Santa

```
santa :: Group -> Group -> IO ()
santa elf_gp rein_gp = do
    (task, (in_gate, out_gate)) <- atomically (orElse
                     (chooseGroup rein_gp deliverPresents)
                     (chooseGroup elf_gp meetInStudy))
    operateGate in_gate
    task
    operateGate out_gate
  where
    chooseGroup :: Group -> (IO ())-> STM (IO (), (Gate,Gate))
    chooseGroup gp task = do gates <- awaitGroup gp
                             return (task, gates)
```

Read this.

- http://chimera.labs.oreilly.com/books/1230000000929/pr01.html
- “Parallel and Concurrent Programming in Haskell”
- It covers all these topics and more.
- Also read:
- http://learnyouahaskell.com/
- http://book.realworldhaskell.org/

Bibliography

- Mordechai Ben-Ari. How to solve the Santa Claus problem. *Concurrency: Practice and Experience*, 10(6):485–496, 1998.
- JA Trono. A new exercise in concurrency. *SIGCSE Bulletin*, 26:8–10, 1994.
- Nick Benton. Jingle bells: Solving the Santa Claus problem in Polyphonic C#. Technical report, Microsoft Research, 2003.
- Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In CAR Hoare, M Broy, and R Steinbrueggen, editors, *Engineering Theories of Software Construction, Marktoberdorf Summer School 2000*, NATO ASI Series, pages 47–96. IOS Press, 2001.
- Anthony Discolo, Tim Harris, Simon Marlow, Simon Peyton Jones, and Satnam Singh. Lock-free data structures using STMs in Haskell. In *Eighth International Symposium on Functional and Logic Programming (FLOPS'06)*, April 2006
- Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'05), June 2005.
- Tim Harris and Simon Peyton Jones. Transactional memory with data invariants. In First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06), Ottowa, June 2006. ACM.
- James Larus and Ravi Rajwar. Transactional memory. Morgan & Claypool, 2006.
- Edward A. Lee. The problem with threads. IEEE Computer, 39(5):33–42, May 2006.