STM

Barbu Paul - Gheorghe

The Haskell implementation

```
-- The STM monad itself
data STM a
instance Monad STM
  -- Monads support "do" notation and sequencing

-- Exceptions
throw :: Exception -> STM a
catch :: STM a -> (Exception->STM a) -> STM a

-- Running STM computations
atomic :: STM a -> IO a
retry  :: STM a
orElse :: STM a -> STM a -> STM a

-- Transactional variables
data TVar a
newTVar   :: a -> STM (TVar a)
readTVar  :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

Figure 1: The STM interface

Why?

Much simpler programming model - no need for locks

Alternative to “conventional” synchronization techniques

Seems to work for DBs

How?

All or nothing approach

=> Optimistic synchronization

Mechanism: Transaction Log

```haskell
type Account = T.TMVar Float

openAccount :: Float -> T.STM (Account)
openAccount balance = T.newTMVar balance

transfer :: Account -> Account -> Float -> T.STM Float
transfer accountA accountB amount = do
    startingBalanceA <- T.takeTMVar accountA
    startingBalanceB <- T.takeTMVar accountB

    let finalBalanceA = (startingBalanceA - amount)
    let finalBalanceB = (startingBalanceB + amount)

    T.putTMVar accountA finalBalanceA
    T.putTMVar accountB finalBalanceB

    return $ amount

main :: IO ()
main = do
    accountA <- T.atomically (openAccount 20)
    accountB <- T.atomically (openAccount 50)
    amt <- T.atomically (transfer accountA accountB 30)
    print $ amt
```

Benefits

Easier to understand

Composable (sequential and alternative)

```haskell
withdraw :: TVar Int -> Int -> STM ()
withdraw acc n = do
    bal <- readTVar acc
    if bal < n then retry
    writeTVar acc (bal-n)

main :: IO ()
main = atomically $ do
    withdraw a1 3
    withdraw a2 7
```

```
atomic (a1 > 3 && a2 > 7) {
    withdraw(a1, 3);
    withdraw(a2, 7);
}
```

Disadvantages

```
atomic {
    if (n>k) then launch_missiles(); S2
}
```

… the type system solves this in Haskell, other languages are still exposed

Transaction log overhead

```haskell
runPhilosopher :: String -> (Fork, Fork) -> IO()
runPhilosopher name (left, right) = forever $ do
    putStrLn $ name ++ " is hungry"

    (leftNum, rightNum) <- atomically $ do
        leftNum <- takeFork left
        rightNum <- takeFork right
        return (leftNum, rightNum)

    putStrLn $ name ++ " got forks " ++ show leftNum ++ " and " ++ show rightNum ++ " and is eating"
    delay <- randomRIO (1, 10)
    threadDelay (delay * 1000000)
    putStrLn $ name ++ " is done eating. Going back to thinking."

    atomically $ do
        releaseFork left leftNum
        releaseFork right rightNum

    delay <- randomRIO (1, 10)
    threadDelay (delay * 1000000)
```

```
def dine(self):
  fork1, fork2 = self.forkOnLeft, self.forkOnRight

  while self.running:
      fork1.acquire(True)
      locked = fork2.acquire(False)
      if locked: break
      fork1.release()
      print '%s swaps forks' % self.name
      fork1, fork2 = fork2, fork1
  else:
      return

  self.dining()
  fork2.release()
  fork1.release()
```

Lock checks are explicit

One more lock and the code complexity increases

Bibliography

Composable Memory Transactions - 18th August 2016, Tim Harris, Simon Marlow, Simon Peyton Jones, Maurice Herlihy - Microsoft Research, Cambridge

Haskell and Transactional Memory - April 2010, Simon Peyton Jones - Microsoft Research, Tokyo Haskell Users Group