Concurrency

- How is it different from parallelism?
    - Concurrency is two tasks being performed at the same time, but may result in one being paused while the other runs
    - Parallelism requires at least two processes to be run at a particular moment in time i.e. in parallel (on different processors)
- Concurrency could be non-deterministic
- **Explicitly threaded with inter-thread communication**

  The basic primitives are very simple. To spawn a new thread of execution:

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

```
main = do
  forkIO (forever $ putChar 'o')
  forkIO (forever $ putChar 'O')
```

-
    - These are GHC threads, not OS threads - lightweight and practical to use thousands
    - Returned ThreadID can be used to check status, send an exception etc

  Basic thread functions:

```
forkIO :: IO () -> IO ThreadId
killThread :: ThreadId -> IO ()
threadDelay :: Int -> IO ()
```

-
    - Create thread, kill thread, delay thread for given number of milliseconds
    - Correct version of previous example...

      The previous example wasn't actually right. It should be:

```
main = do
  forkIO (forever $ putChar 'o')
  forkIO (forever $putChar 'O')
  threadDelay (10^6)
```

- **Need interthread communication - Channel (Unbounded FIFO)**
    - Can write whenever you want, Reading will block until there is something to be read

```
newChan :: IO (Chan a)

writeChan :: Chan a -> a -> IO ()
```

```
readChan :: Chan a -> IO a

getChanContents :: Chan a -> IO [a]

isEmptyChan :: Chan a -> IO Bool
dupChan :: Chan a -> IO (Chan a)
```

- **Threads can then communicate using these channels..**

```haskell
main = do
  hSetBuffering stdout NoBuffering
  c <- newChan
  forkIO (worker c)
  forkIO (forever $ putChar '*')
  readChan c

worker :: Chan Bool -> IO ()
worker c = do
  mapM putChar "Printing all the chars"
  writeChan c True
```

  - `Pr***i*n*t*i*n*g* *a*l*l* *t*h*e* *c*h*a*r*s****************`
  - In this example, thread for the worker function is created, the main then creates a thread to indefinitely write '*'
  - Main then tries to read from the channel given to worker, and blocks until the worker thread has printed all of the characters and written True to the channel
  - Then the main thread reads True and finishes.
- Channels create a nice way for threads to communicate
- **Can create deadlocks**
- Typically used in…
  - **Servers** (thread per connection)
  - **Background processes** (where data computed by a thread becomes available incrementally)
- **Not the basic communication primitive in haskell**
  - **MVar**

```haskell
newEmptyMVar :: IO (Mvar a)
takeMVar :: Mvar a -> IO a
putMVar :: MVar a -> a -> IO ()
```

  - • An MVar can be either empty or full.
    • takeMVar will block when it's empty
    • putMVar will block when it's full
  - Uses of MVar
    - As a mutex for some shared state…
    - Or a one-item channel
    - Or to create an idea of shared state
    - Or to build larger abstractions **(like Chan)**
  - **Building Channels from MVars**
    - Tricky
    - readChan needs to block if the channel is empty…
    - So **build it as a linked-list of MVars**

```haskell
type Stream a = MVar (Item a)
data Item a = Item a (Stream a)
data Chan a = Chan (MVar (Stream a)) -- read pointer
                   (MVar (Stream a)) -- write pointer
```

      - Recursive definition of Item lets Stream (through Item) hold: a MVar a MVar a…..

- Then Chan holds two recursive MVar lists, one for reading, one for writing

```haskell
newChan :: IO (Chan a)
newChan = do
  hole <- newEmptyMVar
  readVar  <- newMVar hole
  writeVar <- newMVar hole
  return (Chan readVar writeVar)
```

- Creating a new channel
- The hole itself is represented by an empty MVar and it is placed within both pointers of the Channel
    - **This creates the desired Channel behaviour**
        - as MVars block on reads for empty, the read pointer contains an empty MVar (mirrors Chan behaviour)
        - Empty MVars allow writes, so this fits with the Chan behaviour requirements
    - Same MVar is given to read and to write

```haskell
writeChan :: Chan a -> a -> IO ()
writeChan (Chan _ writeVar) val = do
  newhole <- newEmptyMVar
  oldhole <- takeMVar writeVar
  putMVar oldhole (Item val newhole)
  putMVar writeVar newhole
```

- Writing to a channel
- Easiest to understand using the case of first write…
    - Create a new MVar (newhole)
    - Take the current MVar is writeVar (oldhole, this is the same MVar for readVar on first write)
    - Put the new information and hole into oldhole (this is still the MVar for the read pointer, it now has data in it)
    - Put newhole into the writeVar (empty)
    - **Note: the readVar will just point to one end of the list of MVars, while the writeVar will point to the other end**

```haskell
readChan :: Chan a -> IO a
readChan (Chan readVar ) = do
  stream <- takeMVar readVar
  Item val new <- takeMVar stream
  putMVar readVar new
  return val
```

- Reading from a channel
    - Take the MVar from the readVar (the reader pointer)
    - Take the MVar stored within it (recall item, stream definition, val = information, new = next MVar)

        - Put new into the readVar to move the pointer on
        - Return the information that was within the Item

```haskell
dupChan :: Chan a -> IO (Chan a)
dupChan (Chan _ writeVar) = do
  hole <- takeMVar writeVar
  putMVar writeVar hole
  newReadVar <- newMVar hole
  return (Chan newReadVar writeVar)
```

-
    - Duplicate channel - create a second channel which shares the writeVar but has a separate read pointer
        - Take the MVar in writeVar (hole)
        - Return the value back to writeVar, and then create a new readVar using the value (hole)
        - Return a new Channel with the newReadVar and same writeVar
    - **This will interact badly with our implementation of readChan, since readChan didn't need to return the value to the 'hole'**
        - I.e. we removed the value from readVar and replaced it unnecessarily in our readChan implementation
- Instead of takeMVar, we can use readMVar which is defined as such...

```haskell
readMVar :: MVar a -> IO a
readMVar m = do
  a <- takeMVar m
  putMVar m a
  return a
```

We can fix `readChan` so that it plays nicely with `dupChan`:

```haskell
readChan :: Chan a -> IO a
readChan (Chan readVar _) = do
  stream <- takeMVar readVar
  Item val tail <- readMVar stream
  putMVar readVar tail
```

-
    - ReadChan and DupChan will now work together
- We could implement peeking into channels…

```haskell
unGetChan :: Chan a -> a -> IO ()
unGetChan (Chan readVar ) val = do
  newReadEnd <- newEmptyMVar
  readEnd <- takeMVar readVar
  putMVar newReadEnd (Item val readEnd)
  putMVar readVar newReadEnd
```

- This is superficially OK, but…
- Consider the case "peeking" at an empty channel

-
    - Inserts a value into the readEnd of the readChan

            - Thread 1 reads from the channel, thread 2 does an ungetChan…
                - **Deadlock**

STM - Software Transactional Memory

- Co-ordination in shared-memory concurrent programs requires locks and condition variables generally…
    - Locks are easy to get wrong (**races, deadlocks, error recovery is hard, non-compositional**)
- A program with a small number of locks is manageable but can block lots of threads
    - And adding granular locks makes the program hard to get right
- **Software Transactional Memory tries to solve this**
    - Takes the database concept of transactions
    - **Creates atomic computation**
- Using atomically $ do
    - **Atomic block will commit in an all or nothing way**
    - Isolated execution
    - Cannot deadlock, can generate exceptions
    - **One way to implement this….** 'Optimistic concurrency'
        - Execute the code lock free, log all memory accesses but don't actually execute them, at the end commit the log, retrying blocks on failure
        - This is what is done, driven by isolated execution
            - Code within a transaction is unaware of changes made by any other transaction
            - So on 2 concurrent transaction executions, the first to complete execution is accepted, and the second will see conflict on completion and restart.
    - **But…**
        - We must not touch any transaction variables outside an atomic block
        - We must not have side-effects within an atomic block
    - **Type System saves this headache**
        - Atomically :: STM a -> IO a
            - The STM monad actions have side-effects but are more limited than the IO ones
            - **Mainly reading and writing special transaction variables**
            -
              ```
              newTVar :: a -> STM (TVar a)
              readTVar :: TVar a -> STM a
              writeTVar :: TVar a -> a -> STM ()
              ```
            - **TVar is, semantically, a value container**
                - To be specific they dont have the blocking semantics of MVars
        - Type system won't compile STM actions unless we execute them atomically (or within the STM monad)
    - **STM has other new concepts in it**
        - **Retry**
            - "Abandon and re-execute from the start"
            - Implementation will block on all read variables before retrying

        - We can’t nest uses of atomically
    - **STM also has compositional choice which covers a lot of real cases where we might try that…**

```haskell
orElse :: Stm a -> Stm a -> Stm a

atomically $ do withdraw a1 x `orElse` withdraw a2 x
                         deposit a3 x
```

Since we are tracking TVars we can even include the idea of *invariants*

```haskell
always :: STM Bool -> STM ()
```

Which we can use to maintain a (global) pool of invariants that are checked after each transaction. Transactions that break the invariants are retried

```haskell
newAccount = do
  v <- newTVar 0
  always $ do cts <- readTVar v ; return (cts >= 0)
  return v
```

- **We cannot use arbitrary IO actions within an atomic block**
    - Violates isolated execution and atomicity
    - One atomic thread (threadA) performing IO could read in a value changed by another thread (threadB). ThreadA may succeed using the changed value, while threadB may fail. The system is now wrong.

- **Channels in STM**
    - Let’s consider how to construct communication channels as an example of how STM simplifies concurrency

```haskell
data TChan a

newTChan   :: STM (TChan a)
writeTChan :: TChan a -> a -> STM ()
readTChan  :: TChan a -> STM a
```

We can implement using the same linked-list model we used for MVar based channels:

```haskell
data TChan a = TChan (TVar (TVarList a))
                     (TVar (TVarList a))

type TVarList a = TVar (TList a)
data TList a = TNil | TCons a (TVarList a)
```

- **Now some functions to make these data types do something...**

```haskell
newTChan :: STM (TChan a)
newTChan = do
  hole <- newTVar TNil
  read <- newTVar hole
  write <- newTVar hole
  return (TChan read write)
```

```haskell
readTChan :: TChan a -> STM a
readTChan (TChan readVar _) = do
  listHead <- readTVar readVar
  head <- readTVar listHead
  case head of
    TNil           -> retry
    TCons val tail -> do
        writeTVar readVar tail
        return val
```

Notice how blocking in a read is implemented using `retry`

```haskell
writeTChan :: TChan a -> a -> STM ()
writeTChan (TChan _ writeVar) a = do
  newListEnd <- newTVar TNil
  listEnd <- readTVar writeVar
  writeTVar writeVar newListEnd
  writeTVar listEnd (TCons a newListEnd)
```

-
    - Note that readTChan uses retry if the list is empty - matching channel behaviour of block on empty
    - writeTChan
        - Create a new empty TVar (newListEnd)
        - Read in the current list end
        - Point the write end to the newListEnd empty variable
        - Create a 'pointer' from the previous listend to the new one
- **UngetChan (this was a trouble maker before...)**
    - It introduced the potential for deadlock

```haskell
unGetTChan :: TChan a -> a -> STM ()
unGetTChan (TChan readVar _) a = do
   listHead <- readTVar readVar
   newHead <- newTVar (TCons a listHead)
   writeTVar readVar newHead
```

There are other operations possible too:

```haskell
isEmptyTChan :: TChan a -> STM Bool
isEmptyTChan (TChan read _write) = do
  listhead <- readTVar read
  head <- readTVar listhead
  case head of
    TNil       -> return True
    TCons _ _  -> return False
```

-

        - STMs will just retry -> no deadlocking

```haskell
readEitherTChan :: TChan a -> TChan b -> STM (Either a b)
readEitherTChan a b =
  fmap Left (readTChan a)
    `orElse`
  fmap Right (readTChan b)
```

    -
        - **Utilising the compositional choice of STMs**
- **So STMs look good**
    - Alternative to lock-based concurrency
    - Atomicity, blocking, error handling
- **Drawbacks**
    - **MVar concurrency is faster**
        - **STMs can produce simplified solutions though which may be faster**
    - **Cannot have multi-way communication without abandoning compositionality**
    - **MVar based solutions guarantee fairness**
        - Multiple MVar blocked threads will finally eval in FIFO
        - **We cannot be fair to TVar threads unless we abandon composability**