CS4012 Topics in Functional Programming

Glenn Strong

Concurrency and Software Transactional Memory

Concurrent Haskell

- We have looked at parallel programming
- There is a separate but related approach: Concurrency
- (possibly) non-deterministic
- Explicitly threaded with inter-thread communication
- Control.Concurrent

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

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

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

- You can use the thread ID to check the status of a thread, send it an exception, etc.

- These are not OS threads; GHC threads are lightweight and it's perfectly practical to have thousands of them at once.

Basic thread functions:

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

The program ends when the original (main) thread ends (this is called demonic concurrency)!

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

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

We need some way to communicate between threads. One simple approach is to use a "channel" (a kind of unbounded FIFO)

You can write to the channel whenever you want

Reads from the channel block until something is available

```haskell
newChan :: IO (Chan a)

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

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

getChanContents :: Chan a -> IO [a]

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

Threads can communicate through 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****************
```

Channels provide a nice abstraction that allows threads to communicate effectively – but watch out, you can introduce deadlocks using them.

Typically you might use them in:

- Servers (where you fork a new thread for each connection)
- Background processes (where the data computed by a thread becomes available incrementally)
- etc…

Channels are the communication primitives in languages like Erlang and Go.

In Haskell they are actually built out of something simpler

The basic communication primitive in Haskell is something called an `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

Simple use:

```haskell
main = do
  m <- newEmptyMVar
  forkIO (do putMVar m ’a’; putMVar m ’b’)
  c <- takeMVar m
  print c
  c <- takeMVar m
  print c
```

- We can use an MVar as a mutex for some other shared state
- Or as a one-item channel
- Or for shared state
- Or to build larger abstractions (like Chan).

But building larger abstractions from them can be tricky

If you look at how to build channels you can see the problem

You might want to say:

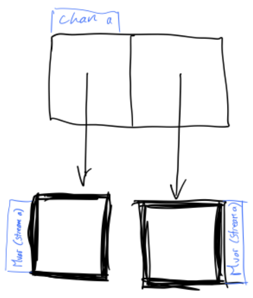
```haskell
data Chan a = MVar [a]
```

(Example from Simon Marlow's discussion of Channels)

Think about how readChan needs to block when the channel is empty

So instead we can build it as a sort of linked-list of MVar's

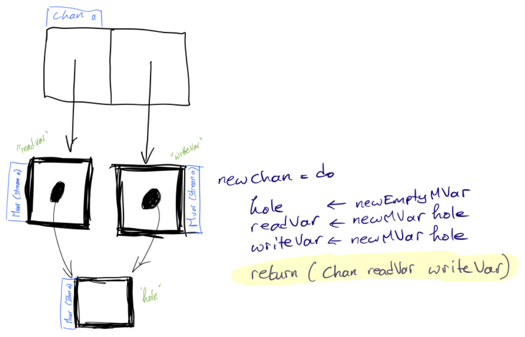
```haskell
type Stream a = MVar (Item a)
data Item a = Item a (Stream a)
data Chan a = Chan (MVar (Stream a)) -- read pointer
                   (MVar (Stream a)) -- write pointer
```



An empty channel contains only the "hole" into which the first item in the stream will be placed.

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

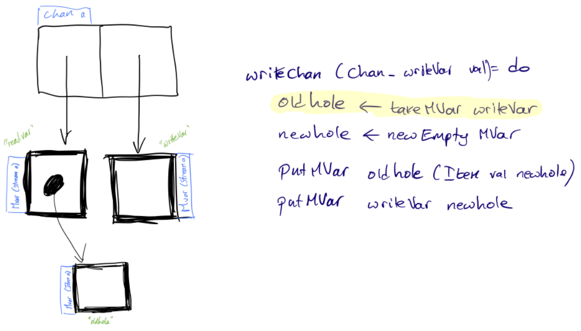
The hole itself is represented by an `MVar`, as are the two end pointers for the linked list.



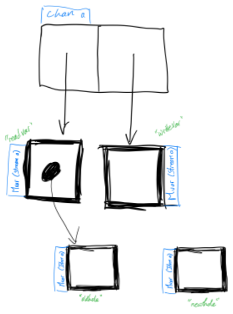
```
newChan = do
  hole     ← newEmptyMVar
  readVar  ← newMVar hole
  writeVar ← newMVar hole
  return (Chan readVar writeVar)
```

Adding an item is quite straightforward

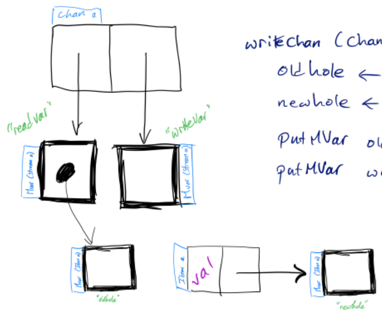
```haskell
writeChan :: Chan a -> a -> IO ()
writeChan (Chan _ writeVar) val = do
  newhole <- newEmptyMVar
  oldhole <- takeMVar writeVar
  putMVar oldhole (Item val newhole)
  putMVar writeVar newhole
```



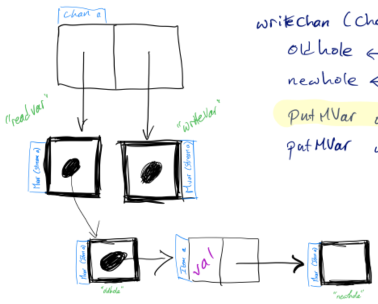
```
writeChan (Chan _ writeVar val) = do
  oldhole ← takeMVar writeVar
  newhole ← newEmpty MVar
  Put MVar  oldhole (Item val newholo)
  put MVar  writeVar  newhole
```



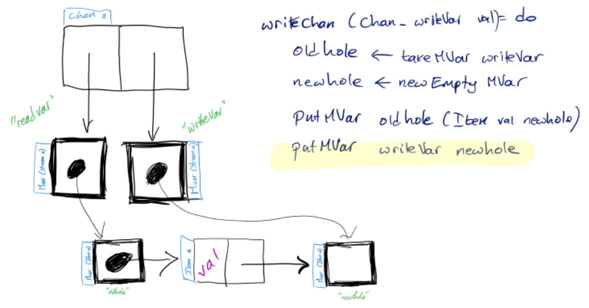
```
writeChan (Chan _ writeVar val)= do
    oldhole ← takeMVar writeVar
    newhole ← newEmptyMVar
    putMVar oldhole (Item val newhole)
    putMVar writeVar newhole
```



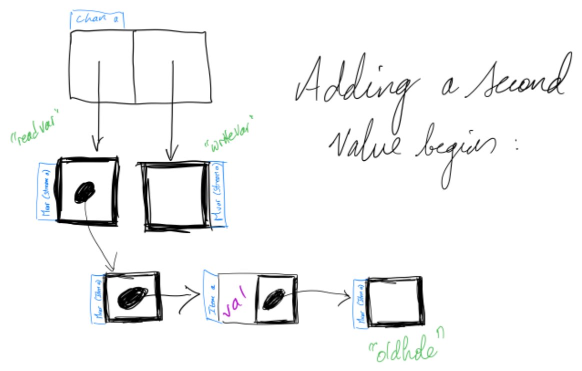
```
writeChan (Chan _ writeVar val)= do
    oldhole ← takeMVar writeVar
    newhole ← newEmptyMVar
    putMVar oldhole (Item val newhole)
    putMVar writeVar newhole
```



```
writeChan (Chan _ writeVar val)= do
    oldhole ← takeMVar writeVar
    newhole ← newEmptyMVar
    putMVar oldhole (Item val newhole)
    putMVar writeVar newhole
```



```
writeChan (Chan _ writeVar val) = do
    oldhole ← takeMVar writeVar
    newhole ← newEmptyMVar
    putMVar oldhole (Item val newhole)
    putMVar writeVar newhole
```



Removing a value ("reading") from the channel is similar:

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

If we think about how the channel blocks on reading we see that we can use this implementation to create *multicast* channels:

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

This operation leaves us with two channels which share their "write" pointer, but have separate "read" pointers.

This definition of `dupChan` will interact badly with our implementation of `readChan`, since `readChan` didn't originally need to return the value to the "hole".

Using the standard `Control.Concurrent` operation `readMVar` (defined thus):

```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
```

Lastly, we might be tempted to implement a form of "peeking" for 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
  - thread 1 reads from the channel
  - and thread 2 attempts to do an `unGetChan`

Software Transactional Memory

- When we talk about co-ordination in shared-memory concurrent programs Locks and Condition variables are the standard technique
- Locks are ridiculously easy to get wrong - Races (when we forget to lock) - Deadlocks (when we lock/release in the wrong order) - Error recovery is hard (e.g. corruption from uncaught exceptions)
- And they are not compositional - We can't build a working system from working pieces

A program with a small number of big locks is usually manageable, but can expect a lot of blocked threads

We could add more granular locks, but then it gets hard to keep the program correct.

Quick example[1]

```
transferFunds a1 a2 amount = do
  withdraw a1 amount
  deposit a2 amount
```

Another thread could observe a time when

- The money has left the first account
- But not arrived in the second account

If we try to be clever:

```
transferFunds a1 a2 amount = do
  lock a1; lock a2
  withdraw a1 amount
  deposit a2 amount
  unlock a2; unlock a1
```

then we might deadlock.

  - The whole thing is a headache, totally non-modular
  - Often we end up wishing we had never heard of this concurrency business.

Need a better idea.

- Recently (mainly post 2005) one has been getting some attention
- Software Transactional Memory
- Steal the idea of "Transactions" from database people

  - This means computations can be done atomically
  - Mix in the idea of pure, first-class functions

In a nutshell:

```
atomically :: Int -> Account -> Account -> IO ()
  transferFunds a1 a2 amount = atomically $ do
  withdraw a1 amount
  deposit a2 amount
```

- The atomic block commits in an all-or-nothing way
- Executes in isolation
- Cannot deadlock, can generate exceptions

- One possible execution strategy ("optimistic concurrency"):
  - Execute code lock-free,
  - logging all memory access instead of performing it
  - At the end, commit the log, retrying the block on failure

[1]See "Beautiful Concurrency", link on BlackBoard

OK, but:

- We must not touch any of the transaction variables outside an atomic block
- We must not have any side-effects inside an atomic block

Type system to the rescue.

```
atomically :: STM a -> IO a
```

- The STM monad actions have side-effects but far more limited ones thatn IO
- Mainly they are about reading and writing special transaction variables

```
atomically :: STM a -> IO a

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

- Semantically TVar values are just containers for values
  - in particular they don't have the blocking semantics of MVar's

We can implement withdraw using this.

```
type Account = TVar Int

withdraw :: Account -> Int -> STM ()
withdraw acc amount = do
  bal <- readTVar acc
  writeTVar acc (bal - amount)
```

The type system keeps us honest:

```
bad :: Account -> IO ()
bad acc = do print "Withdrawing..."
withdraw acc 10
```

- Won't even compile! withdraw is an STM action.

We can fix it:

```
bad acc = do print "Withdrawing..."
  atomically $ withdraw acc 10
```

- This satisfies the first condition (must not touch the affected variables outside the atomic block.

STM has some other new ideas:

```
retry :: STM ()
```

It means "abandon and re-execute from the start"

Implementation will block on all read variables before retrying

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

We cannot nest uses of atomically (what would that even mean?)

STM offers "compositional choice" which covers a lot of the 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 `TVar`s 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
```

Finally, it's probably obvious why we must not allow IO actions inside an atomic block?

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

Fortunately the type checker won't allow this!

```
No instance for (Control.Monad.IO.Class.MonadIO STM)
    arising from a use of 'liftIO'
```

- Versions of STM have been developed (and are used) for C++, Java, C#
- Some difficult to solve problems, though:
  - Retry semantics in impure languages
  - Keeping atomic blocks IO free
  - Keeping the transaction variables untouched
  - Keeping the atomic blocks away from normal variables

Channels in STM

Finally, let's consider how to construct communication channels, as an example of how STM simplifies the concurrency model.

(this implementation is available in `Control.Concurreny.STM.TChan`)

The interface is simple

```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)
```

(we have an explicit `TNil` for the empty list. This was represented by an empty `MVar` before).

```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)
```

Remember how stuck we were trying to implement `unGetChan`?

The `MVar` semantics meant we could never be sure that our program wouldn't deadlock if some thread was using this operation.

```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
```

Exercise: can this be (safely) implemented in the `MVar` version of channels?

How about this:

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

Summary

- STM is a promising alternative to lock-based synchronization for concurrent programming
- it offers *composable*
  - atomicity
  - blocking
  - error handling

There is a price, of course, and it's worth noting it:

- `MVar` based concurrency is generally faster
  - but you can sometimes use `STM` to construct simpler solutions that will be faster then an `MVar` based solution.
- `MVar` based solutions can guarentee *fairness*

  - Multiple `MVar` blocked threads will be woken in FIFO order.
  - We can't be *fair* to `TVar` threads unless we abandon composability

The GHC implementation of STM is very efficient as it is tightly bound into the RTS.

- Each transaction maintains a thread-local *transaction log* recording reads and (tentative) writes.
- Reads consult the log, writes are made only to the log

- When attempting to commit an atomic block the log is checked for validity
  - If the old value of a read variable is pointer-equal to the current TVar value then it passes the validity check
- If all log entries pass then the changes can be applied; this is done without interruption.
- If any validations fail then the log is discarded and a fresh transaction is started with a new log.

- When a `retry` is required the log is discarded and the transaction is added to a wait-queue associated with the TVars in the log
- Any commit that updates the relevant TVars will be unblocked and scheduled to be run again.

Fully portable software-based implementations also exist (see, for example, "A Haskell-Implementation of STM Haskell with Early Conflict Detection" by David Sabel).

- Portable
- But significantly slower than the GHC internal implementation