Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Week 9
Concurrency

CS4012

Topics in Functional Programming
Michaelmas Term 2020

Glenn Strong <glenn.Strong@scss.tcd.ie>

Basics of concurrency

Previously we looked at parallel programming — a way to try to improve performance without changing program meaning.

There is another separate but related idea: Concurrency

- **(possibly) non-deterministic**
- **Explicitly threaded with inter-thread communication**

In Haskell this is delivered via the Control.Concurrent module

Basics of Concurrency

The basic primitives are very simple.

To spawn a new thread of execution use ForkIO

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

It works very much as you'd imagine.

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

Basics of Concurrency

The ThreadID value identifies the running thread.

You can use the ID to check the status of the thread and to send signals to the thread (for example, to shut it down).

In the standard GHC implementation these are *not* OS level threads,.

They are lightweight and it's perfectly practical to have thousands of them in a running program.

Basics of Concurrency

When the *original* thread (the one running Main) terminates then the whole program terminates. Compare this example to the previous one and see how the behaviour differs:

```
main = do
  hSetBuffering stdout NoBuffering
  forkIO $ forever (putChar 'o')
  replicateM_ 10000 $ putChar 'O'
```

Basics of Concurrency

The basic thread interface has only a few functions:

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

Start a thread, getting a new ID, or find the ID of this thread

```
killThread :: ThreadID -> IO ()
```

Terminate a thread

```
threadWait      :: Int -> IO ()
threadWaitRead  :: Fd  -> IO ()
threadWaitWrite :: Fd  -> IO ()
```

Make this thread block (for some microseconds, or until a file descriptor is ready to read or write)

```
yield :: IO ()
```

Yield to another thread

Basics of Concurrency

So we could have made the "Main" thread block for a while in our example instead:

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

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

End of part 1

Glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Week 9, Part 2
Communication & Concurrency

Glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/

Inter-thread Communication

We need some way to communicate between threads.

A simple approach is to use a "channel" (which is a kind of unbounded FIFO)

- You can write to the channel whenever you want
- Reads from the channel block until something is available

As you'd expect from Haskell the channels provided by Control.Concurrent are strongly typed and first order.

Inter-thread Communication

Channel basic API

The basic Channel interface looks like this:

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

Create a new empty channel

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

Write to a channel

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

Read from a channel, blocking while the channel is empty

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

Duplicate a channel (but not the existing contents)

```
getChanContents :: Chan a -> IO [a]
writeList2Chan :: Chan a -> [a] -> IO ()
```

Lazy-read a channel or write a lazy list to a channel

Inter-thread Communication

Channel example

A small example of using channels to communicate:

```
import Control.Concurrent
import Control.Monad
import System.IO

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
```

Inter-thread Communication

Channel comments

Channels provide a nice abstraction that allows threads to communicate effectively. Example uses include:

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

You have to take care when using Channels because races and deadlocks are possible with them.

Channels are not actually communication *primitives* in Haskell (as they are in Go or Erlang). The actual primitives are simpler.

Inter-thread Communication

MVars

The basic communication primitive in Haskell is something called an "MVar".

An MVar is:

- A single-item shared variable
- A box that can be empty or full
- A synchronising variable

Inter-thread Communication

MVar basic API

The basic MVar interface looks like this:

```
newEmptyMVar ::        IO (MVar a)
newMVar      :: a -> IO (MVar a)
```

```
takeMVar :: Mvar a -> IO a
```

```
putMVar  :: MVar a -> a -> IO ()
```

Create a new MVar

Read a value from a full MVar, leaving it empty.

Write a value to an empty MVar, leaving it full.

The clever bit about MVars is their empty/full behaviour.

- **takeMVar will block when the target MVar is empty**
- **putMVar will block when the target MVar is full**

Inter-thread Communication

MVars comments

You can use an MVar as:

- A mutex for some shared state
- A one-item channel
- A binary semaphore (take/put used as wait/signal actions)
- As building blocks for larger abstractions (like Chan)

MVars are far from foolproof.

Race conditions, deadlocks, uncaught exceptions and all the rest lurk here, so we must be cautious when using them.

The Control.Concurrent.MVar module has a number of other utilities, including non-blocking versions of read/write.

Inter-thread Communication

MVar example

A small example of using MVars to communicate:

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

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

End of part 2

Glenn.Strong@scss.tcd.ie

https://scss.tcd.ie/Glenn.Strong/

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Part 3
Building a Channel

Glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/

Inter-thread Communication

Building a Channel

If you look at how to build channels our of MVars it quickly becomes clear that it's not trivial.

You might be tempted by this:

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

But think about how readChan needs to behave when the channel is empty.

Checking whether the list is empty requires you to lock the MVar *first*. The MVar itself is not empty then the channel is, to the MVar semantics won't do it.

Inter-thread Communication

Building a Channel

A better solution is to think of a way to build Channels as a linked-list of MVars

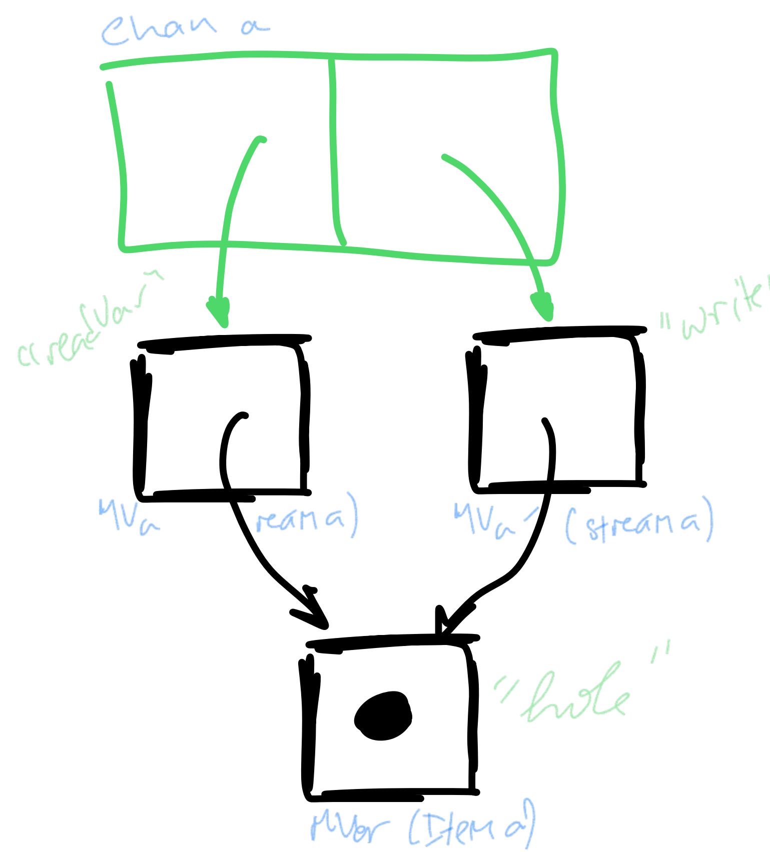
```
type Stream a = MVar (Item a)
data Item a = Item a (Stream a)
data Chan a = Chan (MVar (Stream a)) -- read pointer
                   (MVar (Stream a)) -- write pointer
```

Inter-thread Communication

Building a Channel

An empty channel contains a “hole” into which the first item in the stream will be placed.

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

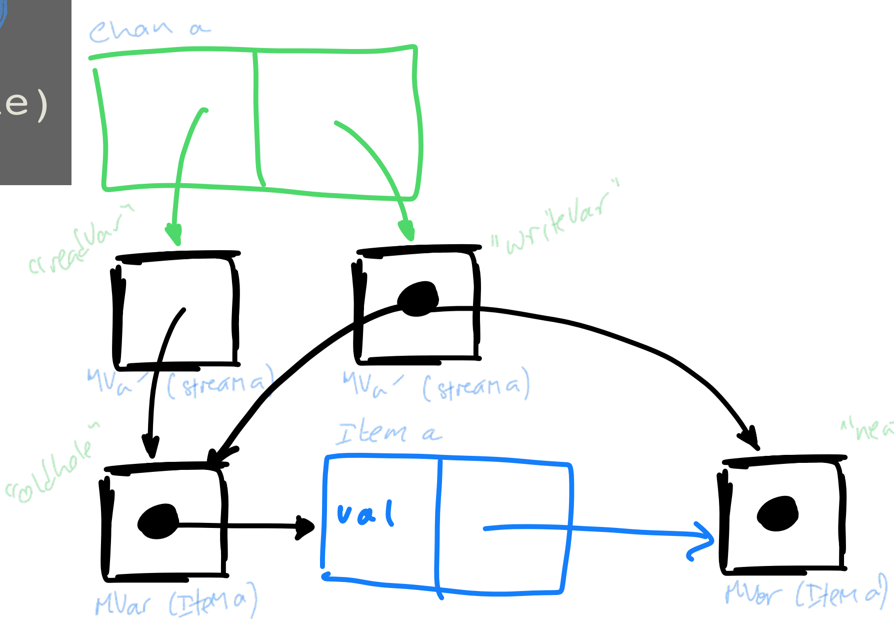


Inter-thread Communication

Building a Channel

Adding an item is straightforward:

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



Inter-thread Communication

Building a Channel

Removing an item follows a similar pattern

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

Inter-thread Communication

Building a Channel

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

```
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.

Inter-thread Communication

Building a Channel

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".

What we would like is an operation that will give us a copy of the contents of an MVar but leave the value in the MVar as well.

We actually have this in the Control.Concurrent.MVar library already:

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

Inter-thread Communication

Building a Channel

Using this definition we can fix readChan so that it plays nicely with dupChan:

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

The real point here is that working with MVars can be subtle, and if we're not careful then we can introduce blocking behaviour all too easily.

Inter-thread Communication

Building a Channel

Another operation we might be tempted by would be "peeking" at the front of a channel.

```
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 of "peeking" at an empty channel:

- **Thread 1 reads from the channel**
- **Thread 2 attempts to perform an "unGetChan"**

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

End of part 3

Glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Part 4
Software Transactional Memory

Glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/

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)

Software Transactional memory

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.

Software Transactional memory

Take this example

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

A second thread in the same program could observe a time when both of these are true:

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

Software Transactional memory

The typical way to approach this is via *locks*.

But if we try to be clever, for example:

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

Then we could deadlock the program!

```
do
  forkIO $ transferFunds acc1 acc2 100
  forkIO $ transferFunds acc2 acc1 12
```

Software Transactional memory

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

Software Transactional memory

In a nutshell:

```
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**

Software Transactional memory

How could you execute such a thing safely?

One possible execution strategy:

- **Execute code lock-free,**
- **logging all memory access instead of performing it**
- **At the end, lock everything and commit the log**
  - **Retrying the block on failure**

Software Transactional memory

OK, but…

We have to ensure that the variables involved in the transaction are not touched *outside* of an atomic block

We have to ensure there are no side-effects *inside* an atomic block.

In other words, we need to partition our program into “atomically safe” and “atomically unsafe” regions.

Software Transactional memory

Type system to the rescue!

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

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

Software Transactional memory

STM basic API

The basic STM interface looks like this:

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

Run an STM action to completion

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

Create a variable that can be used in an STM action

```
readTVar :: TVar a -> STM a
```

Get the value of a TVAR

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

Change the value of a TVAR

```
retry  :: STM ()
orElse :: STM a -> STM a -> STM a
```

Force a retry or
Give an alternative path on retry

Software Transactional memory

We can use STM to try to solve the problem in our example:

```
type Account = TVar Int

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

This is now an STM action, not an IO action

Software Transactional memory

The type system keeps us honest

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

Nope, types don’t line up, ‘withdraw’ is not an IO action.

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

This satisfies the first condition (actions must not touch the affected transaction variables outside the atomic block)

Software Transactional memory

The atomic block abandon and retry if another thread interferes with the TVars in this block.

We can also force a retry if we detect a condition that requires it.

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

For example:

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

STM will block on all read variables before retrying

Software Transactional memory

Finally: 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:

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

For example:

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

(semantically: try the first action, if it fails try the second, if *that* fails then retry the whole thing)

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

End of part 4

Glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Thank you

glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/