Transactions of Actions

Retry on the Fly with Software Transactional Memory

Laurens Duijvesteijn

June 2018

duijf.io · github.com/duijf · @_duijf

Software Transactional Memory

"Garbage collection-esque" concurrency control

Software Transactional Memory

“Garbage collection-esque” concurrency control

But that’s getting ahead of ourselves

Before we start...

Dabbled with Haskell since 2014

Love to talk about it and introduce it to people

Dev at Channable, a product advertising startup. We have been using Haskell in production for 1.5 years

Job scheduling

CLI tooling

Github: `channable/vaultenv`

Reverse proxy/ingress

Websocket-enabled document store

Github: `channable/icepeak`

Data processing

Some STM sprinkled around

However, my interest is largely personal

Who are you?

What do you hope to learn?

What’s your (Haskell) background?

Goals

What are the problems that STM solves?

How (and when) can STM help you?

Discuss some implementation notes

Structure

Mix of me talking and you working

I talk a bit, then you code a bit

Structure

Mix of me talking and you working

I talk a bit, then you code a bit

I might do some interactive stuff
or let you present your findings

Structure

Mix of me talking and you working

I talk a bit, then you code a bit

I might do some interactive stuff
or let you present your findings

Also: Wednesday isn't set in stone.
I'll try to cover what's interesting to you.

Lay of the land

Concurrent

Parallel

Concurrent

A property of the program

Parallel

Concurrent

A property of the program

Parallel

A property of the machine

Concurrent

Steps can happen at the same time

Parallel

Steps actually happen at the same time

Concurrent

A matter of potential

Parallel

Realization of potential

Why care about concurrent?

Why care about concurrent?

Speed. (On multicore-machines)

Why care about concurrent?

Speed. (On multicore-machines)

There's also UX, fault tolerance, etc..

We need to invest in concurrent programs

We live in a post-Moore world. Hitting limits of power and size

We need to invest in concurrent programs

We live in a post-Moore world. Hitting limits of power and size

If we're going to write faster programs, we need to care about multicore.

The Perils of Potential

Most programs use and transform data

Most programs use and transform data

A configuration of data in a program is a state

Our view of the world depends on program state

Our view of the world depends on program state

Can be spread accross multiple cores and threads

We want

Consistent views

Atomic updates

Isolated transactions

So what do we do with state?

So what do we do with state?

That depends on our concurrency model

Shared Memory

Message Passing

Shared Memory

Communicate by sharing

Message Passing

Shared Memory

Communicate by sharing

Message Passing

Share by communicating

Today, we’re interested in sharing and mutation

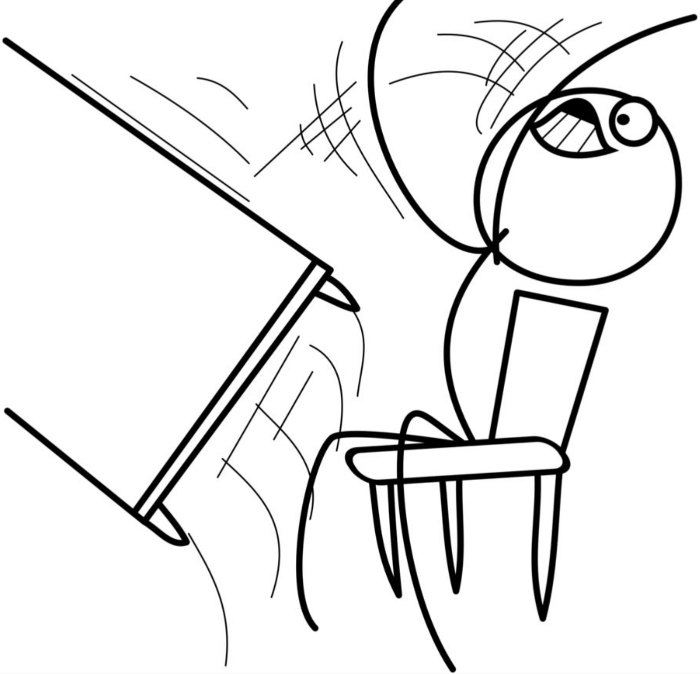
… and mutation is difficult and annoying

A Yak Shaving excersise

```
fn Transfer(Account from, Account to, Decimal amount) {
  to.Credit(amount);
  from.Debit(amount);
}
```

```
fn Transfer(Account from, Account to, Decimal amount) {
  to.Credit(amount);
  from.Debit(amount);
}
```

```
fn Transfer(Account from, Account to, Decimal amount) {
  lock(from); lock(to);
  to.Credit(amount);
  from.Debit(amount);
  unlock(from); unlock(to);
}
```



```
Thread 1                        Thread 2
   .                               .
   .                               .
Transfer(foo, bar, 100);    Transfer(bar, foo, 40);
   .                               .
   X                               X
```

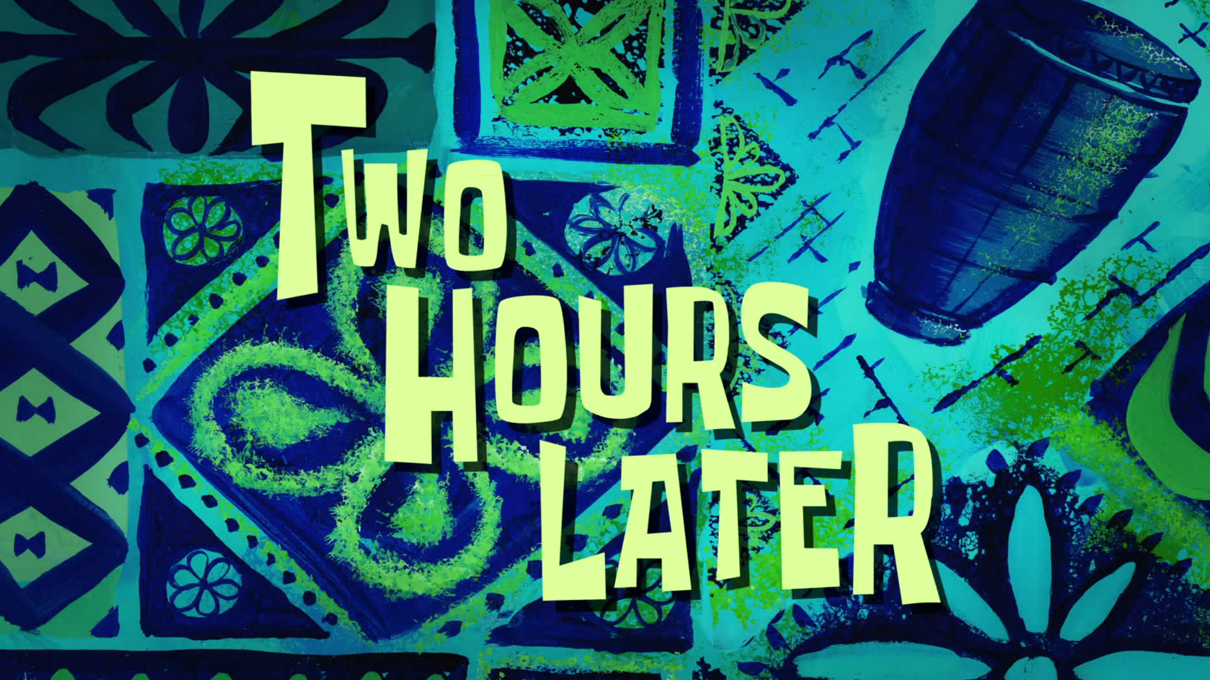
```
fn Transfer(Account from, Account to, Decimal amount) {
  // Decide on locking order
  first = ...; second = ...;

  lock(first); lock(second);
  to.Credit(amount);
  from.Debit(amount);
  unlock(first); unlock(second);
}
```

Other source of difficulty…

Actually remembering to lock everywhere

Atomicity with error handling/exceptions


TWO HOURS LATER

Assume we now have a perfect `Transfer()`

How do we use it?

In a nested transaction with complex logic?

In a nested transaction with complex logic?

Without causing problems?

In a nested transaction with complex logic?

Without causing problems?

And without knowing about it’s internals?

You can't.

You need to have knowledge of it’s internals

Because of this, you control flow gets flipped…

Concurrent code is where composition goes to die.

Concurrent code is where composition goes to die.

encapsulation goes to die.

Concurrent code is where

your sanity goes to die.
composition goes to die.
encapsulation goes to die.

Light at the end of the tunnel

There must be a better way

There must be a better way

Most of us don't use `malloc()` and `free()`

Garbage collectors are a thing and are used succesfully

Can we let the computer take care of locks?

Can we let the computer take care of locks? What would that look like?

```
fn Transfer(Account from, Account to, Decimal amount) {
  atomically {
    to.Credit(amount);
    from.Debit(amount);
  }
}
```

Where code in the `atomically` block has all the properties we want

Where code in the `atomically` block has all the properties we want

Let’s switch to code that you can actually run…

```haskell
-- STM () means: an STM action with no result
transfer :: Account -> Account -> Decimal -> STM ()
transfer from to amount = do
  credit to   amount
  debit  from amount
```

```haskell
import Control.Concurrent.STM
import Data.Decimal

type Account = ...

main :: IO ()
main = do
  foo <- ...
  bar <- ...
  -- atomically :: STM a -> IO a
  -- transfer foo bar 300 :: STM ()
  atomically (transfer foo bar 300)
```

```haskell
import Control.Concurrent.STM
import Data.Decimal

type Account = TVar Decimal

main :: IO ()
main = do
  foo <- ...
  bar <- ...
  -- atomically :: STM a -> IO a
  -- transfer foo bar 300 :: STM ()
  atomically (transfer foo bar 300)
```

```haskell
import Control.Concurrent.STM
import Data.Decimal

type Account = TVar Decimal

main :: IO ()
main = do
  foo <- newTVarIO 4242
  bar <- newTVarIO 5000
  -- atomically :: STM a -> IO a
  -- transfer foo bar 300 :: STM ()
  atomically (transfer foo bar 300)
```

Semantics of `atomically`

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

Semantics of `atomically`

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

External observers never view intermediate states

Transactions happen succesfully if there aren't any conflicting changes

Retry a transaction if there are conflicts

How do we know about conflicts?

Keep an access log where we record reads and writes

Before a block inside `atomically` commits, check the log of all involved `TVar`s for conflicts

How do we know about conflicts?

Keep an access log where we record reads and writes

Before a block inside `atomically` commits, check the log of all involved `TVar`s for conflicts

t_1 and t_2 conflict when:

- Their write sets overlap
- The write set of t_1 overlaps with the read set of t_2
- The write set of t_2 overlaps with the read set of t_1

How do we retry a transaction?

We jump to the start of the transaction and try again, until we succeed.

How do we retry a transaction?

We jump to the start of the transaction and try again, until we succeed.

This helps for conflicts (if there isn’t a lot of contention)

This is also the error handling mechanism within STM.

```haskell
debit :: Account -> Decimal -> STM ()
debit account amount = do
  balance <- readTVar account
  writeTVar account (balance - amount)

-- Credit is the same, but with + instead of -
```

```haskell
debit :: Account -> Decimal -> STM ()
debit account amount = do
  balance <- readTVar account
  if balance - amount < 0
    then retry
    else writeTVar account (balance - amount)
```

`retry` semantics

Retry the entire transaction from the start*

This ensures we don’t have to undo all our previous work to remain consistent.

`retry` semantics

The runtime retries only when some of the inputs change, to avoid busy wait.

`retry` semantics

The runtime retries only when some of the inputs change, to avoid busy wait.

Sometimes we don't want to retry perpetually

To avoid perpetual retries, use `orElse :: STM a -> STM a -> STM a`, which augments retry semantics.

To avoid perpetual retries, use `orElse :: STM a -> STM a -> STM a`, which augments retry semantics.

Or return a value out of a transaction to indicate success/failure. (You can get it out with `<-`)

```haskell
transfer :: Account -> Account -> Decimal -> STM ()
transfer from to amount = actualTransfer `orElse` noOp
  where
    actualTransfer = do
      debit  from amount
      credit to   amount
    noOp = pure ()
```

```haskell
import Control.Applicative

transfer :: Account -> Account -> Decimal -> STM ()
transfer from to amount = actualTransfer <|> noOp
  where
    actualTransfer = do
      debit  from amount
      credit to   amount
    noOp = pure ()
```

Final words before we code

What STM gets us

Atomic transactions for shared memory

Sane control flow (no flipping inside out)

Encapsulation of concurrent code

Helps avoid common locking problems

STM works...

By keeping a transaction log, retrying on conflicts

Using three basic combinators: `atomically`, `retry` and `orElse`

On a variety of datatypes. We've seen `TVar`, but there's `TChan`, `TQueue`, etc..

Just like garbage collectors, STM is no silver bullet.

Writing concurrent programs is still difficult, but STM can take away some of the pain.

Just like garbage collectors, STM is no silver bullet.

Writing concurrent programs is still difficult, but STM can take away some of the pain.

Example: starvation/contention of long running transactions

Time for some work

https://bit.ly/2MgRVEo