Introduction to Software Transactional Memory

For Haskell Amsterdam

Laurens Duijvesteijn

June 2020

duijf.io · github.com/duijf · hi@duijf.io

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

Currently: teamlead of the DevOps team at Channable. We have been using Haskell in production since 2017.

Job scheduling

CLI tooling

Reverse proxy/ingress

Websocket-enabled document store

Data processing

Merge bot

Email sending service

Tech blog: `tech.channable.com`

Github: `github.com/channable`

Hiring: `jobs.channable.com`

Channable has some STM sprinkled around

My interest is largely personal

Goals

What are the problems that STM solves?

How (and when) can STM help you?

Discuss some STM datatypes

Not: discuss the runtime implementation

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);
}
```

That code is not thread safe.

```
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);
}
```

Still incorrect.

```
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

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 is turned ’inside out’

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 `TVars` 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 `TVars` 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`

To avoid perpetual retries:

Use `orElse :: STM a -> STM a -> STM a`

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 ()
```

Recap

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..

More datatypes

`TVars` are nice

But sometimes, we want something more

`TVars` are nice

But sometimes, we want something more

Certain situations call for more datatypes

`TVars` are nice

But sometimes, we want something more

Certain situations call for more datatypes

The `stm` package provides a bunch of them

Stuff in `stm`

`TVar` Variables

Stuff in `stm`

`TVar` Variables

`TArray` Arrays (not discussing these)

Stuff in `stm`

`TVar` Variables

`TArray` Arrays (not discussing these)

`TMVar` Variable which is either empty or full

Stuff in `stm`

`TVar` Variables

`TArray` Arrays (not discussing these)

`TMVar` Variable which is either empty or full

`TQueue` Queues (bounded: `TBQueue`)

`TChan` Channels (slower than queues, but support broadcast)

TMVar

Introducing `TMVar a`

Empty or filled (name comes from `MVar`, more later)

Can be used as a synchronization primitive

```
newTMVar :: a -> STM (TMVar a)
newEmptyTMVar :: STM (TMVar a)

putTMVar  :: TMVar a -> a -> STM () -- blocks
takeTMVar :: TMVar a -> a -> STM a  -- blocks
```

Can we build `TMVar` ourselves?

Yes.

Can we build `TMVar` ourselves?

Yes. In Haskell, even.

Can we build `TMVar` ourselves?

Yes. In Haskell, even. It's basically a `TVar (Maybe a)`

Can we build `TMVar` ourselves?

Yes. In Haskell, even. It's basically a `TVar (Maybe a)`

Really. Add a `newtype` and that's the `stm` implementation

```haskell
newtype TMVar a = TMVar (TVar (Maybe a))

newTMVar :: a -> STM (TMVar a)
newTMVar contents = do
  inner <- newTVar (Just contents)
  pure (TMVar inner)
```

```haskell
newtype TMVar a = TMVar (TVar (Maybe a))

newEmptyTMVar :: STM (TMVar a)
newEmptyTMVar = do
  inner <- newTVar Nothing
  pure (TMVar inner)
```

```haskell
putTMVar :: TMVar a -> a -> STM ()
putTMVar (TMVar inner) newContents = do
  contents <- readTVar inner
  case contents of
    Nothing -> do
      writeTVar inner (Just newContents)
      pure ()
    Just _ -> retry
```

```haskell
takeTMVar :: TMVar a -> STM a
takeTMVar (TMVar inner) = do
  contents <- readTVar inner
  case contents of
    Nothing -> retry
    Just c -> do
      writeTVar inner Nothing
      pure c
```

`MVar`

`TMVar`

`MVar`

Defined in `Control.Concurrent.MVar`

`TMVar`

Defined in `Control.Concurrent.STM.TMVar`

`MVar`

Concurrent Haskell (1996)

`TMVar`

Composable Memory Transactions (2006)

`MVar`

(First come, first serve) fairness

`TMVar`

No fairness guarantees

`MVar`

Lower level tool. Can be used against starvation

`TMVar`

High level convenient STM interface

TQueue

Introducing `TQueue a`

Unbounded, first-in first-out, queue

$O(1)$ inserts and reads (amortized).

```haskell
newTQueue :: STM (TQueue a)

writeTQueue :: TQueue a -> a -> STM ()
readTQueue :: TQueue a -> STM a -- blocks
```

Can we build `TQueue a` ourselves?

Again, yes. In this case using two `TVar [a]`s

Can we build `TQueue a` ourselves?

Again, yes. In this case using two `TVar [a]`s

```
data TQueue a
  = TQueue (TVar [a]) -- waiting to be read
           (TVar [a]) -- written
```

```haskell
data TQueue a = TQueue (TVar [a]) (TVar [a])

newTQueue :: STM (TQueue a)
newTQueue = do
  read  <- newTVar []
  write <- newTVar []
  pure (TQueue read write)
```

```haskell
data TQueue a = TQueue (TVar [a]) (TVar [a])

writeTQueue :: TQueue a -> a -> STM ()
writeTQueue (TQueue _read write) a = do
  listend <- readTVar write
  writeTVar write (a:listend)
```

```haskell
data TQueue a = TQueue (TVar [a]) (TVar [a])

readTQueue :: TQueue a -> STM a
readTQueue (TQueue read write) = do
  xs <- readTVar read
  case xs of
    (x:xs') -> do
      writeTVar read xs'
      pure x
    [] -> do
      --
      -- What should go here??
      --
```

```haskell
data TQueue a = TQueue (TVar [a]) (TVar [a])

readTQueue :: TQueue a -> STM a
readTQueue (TQueue read write) = do
  xs <- readTVar read
  case xs of
    (x:xs') -> do
      writeTVar read xs'
      pure x
    [] -> do
      -- Get the contents of `write`, reverse, and
      -- store in `read`. (retry if `write` is empty).
      -- This is where the "amortized" came from.
```

Why add bounds?

Bounded queue: configurable (fixed) capacity. Writes block (`retry` with STM) if the new size is larger than the capacity.

Useful for implementing backpressure in our systems.

How do we add bounds?

How do we add bounds?

We need to know the sizes of the `read` and `write` lists.

How do we add bounds?

We need to know the sizes of the `read` and `write` lists.

For every insert, we can see if we exceed capacity.

```haskell
data TBQueue a
  = TBQueue (TVar Int) -- read capacity
            (TVar [a]) -- waiting to be read
            (TVar Int) -- write capacity
            (TVar [a]) -- written
```

```haskell
newTBQueue :: Int -> STM (TBQueue a)
newTBQueue size = do
  read  <- newTVar []
  write <- newTVar []
  rsize <- newTVar 0
  wsize <- newTVar size
  pure (TBQueue rsize read wsize write)
```

```haskell
writeTBQueue :: TBQueue a -> a -> STM ()
writeTBQueue (TBQueue rsize _read wsize write) a = do
  w <- readTVar wsize
  if (w /= 0)
    then do writeTVar wsize $! w - 1
    else do
          --
          --
          -- What should go here?
          --
          --
  listend <- readTVar write
  writeTVar write (a:listend)
```

```haskell
writeTBQueue :: TBQueue a -> a -> STM ()
writeTBQueue (TBQueue rsize _read wsize write) a = do
  w <- readTVar wsize
  if (w /= 0)
    then do writeTVar wsize $! w - 1
    else do
          -- Figure out if we have read capacity.
          -- Retry if not.
          --
          -- If we have some, swap the read and
          -- write capacities and subtract 1.
  listend <- readTVar write
  writeTVar write (a:listend)
```

```haskell
writeTBQueue :: TBQueue a -> a -> STM ()
writeTBQueue (TBQueue rsize _read wsize write) a = do
  w <- readTVar wsize
  if (w /= 0)
    then do writeTVar wsize $! w - 1
    else do
          r <- readTVar rsize
          if (r /= 0)
            then do writeTVar rsize 0
                    writeTVar wsize $! r - 1
            else retry
  listend <- readTVar write
  writeTVar write (a:listend)
```

stm-containers

Sometimes you want an STM (hash)map

This can be useful whenever a Hashmap is useful

Example: user-ID to rate limiting information.

A first idea might be `TVar (HashMap (TVar a))`, but that's not ideal.

Why not `TVar (HashMap (TVar a))`?

Updates to invidivual keys are fine.

Updates to the entire `HashMap` change the outer `TVar`

This imposes a bottleneck for some workloads (contention on key addition/removal)

We want to decrease contention

This requires splitting the workload into some chunks.

It does basically mean “implement `Map` again”.

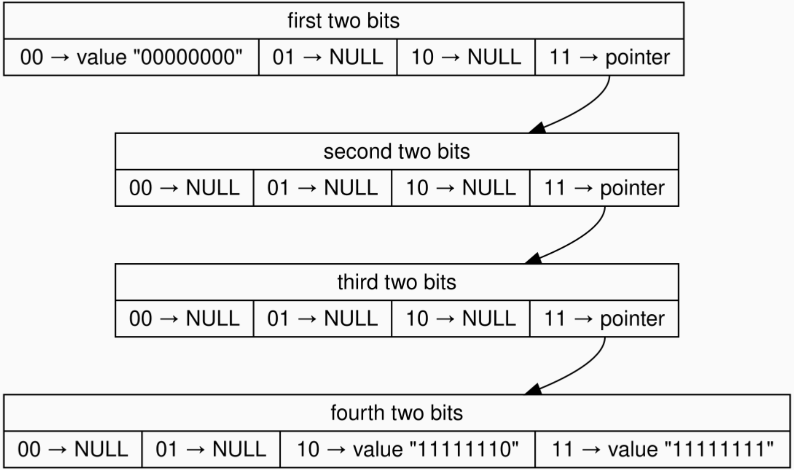
Method: a “Hash Array Mapped Trie”

HAMT, briefly

Each key/value-pair is stored by:

1. Hashing the key (yielding a binary value)
2. Storing the value in the Trie at the location determined by the hash

More at: https://idea.popcount.org/2012-07-25-introduction-to-hamt/



Picture credit: https://idea.popcount.org/2012-07-25-introduction-to-hamt/

You can imagine this is better

You can imagine this is better

Updates to a part of the Map are almost isolated

You can imagine this is better

Updates to a part of the Map are almost isolated

We sometimes need to insert new levels, but this is waaay better than global contention.

You can imagine this is better

Updates to a part of the Map are almost isolated

We sometimes need to insert new levels, but this is waaay better than global contention.

A real implementation is more involved and has compression and an bunch of other things.

More at: https://nikita-volkov.github.io/stm-containers/

Summary

Summary

Just like garbage collectors, STM is no silver bullet.

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

Summary

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
Also: having to keep your queues bounded

Summary

`TMVar a = TVar (Maybe a)`

`TQueue a = TQueue (TVar [a]) (TVar [a])`

Want bounds? Add counts for read and write capacity!

`stm-containers` works with a Hash Array Mapped Trie to avoid contention on structural updates

Next

We didn't discuss the implementation in the runtime

Neither did we get into the operational semantics yet

Paper suggestion: "Composable Memory Transactions"

Laurens Duijvesteijn

`github.com/duijf`

`duijf.io`

`hi@duijf.io`