Introduction to STM

Part II: Transactional LEGOs

Laurens Duijvesteijn

June 2018

`duijf.io` · `github.com/duijf` · `@_duijf`

Preface

Recap (1/2)

STM = Optimistic concurrency control

Recap (1/2)

STM = Optimistic concurrency control

A `TVar` is a transactional variable

Recap (1/2)

STM = Optimistic concurrency control

A `TVar` is a transactional variable

Operations on `TVar`s are called STM transactions

Recap (1/2)

STM = Optimistic concurrency control

A `TVar` is a transactional variable

Operations on `TVar`s are called STM transactions

Run transactions with `atomically`; combine with `orElse`

Recap (2/2)

Transactions get rerun on conflicts. We know because of a log

Recap (2/2)

Transactions get rerun on conflicts. We know because of a log

You can also call `retry` to rerun yourself

Recap (2/2)

Transactions get rerun on conflicts. We know because of a log

You can also call `retry` to rerun yourself

The runtime reruns the transaction after its inputs change

Recap (2/2)

Transactions get rerun on conflicts. We know because of a log

You can also call `retry` to rerun yourself

The runtime reruns the transaction after its inputs change

Use this for blocking operations

Today (by request)

Using `TVar` as a piece of LEGO

Let's create some of the datatypes in `stm`

If we have time, we can look at `stm-containers`

Note: this is all other people's code

Transactional LEGOs

`TVars` are nice

But sometimes, we want something more

`TVar`s 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.

```
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”.

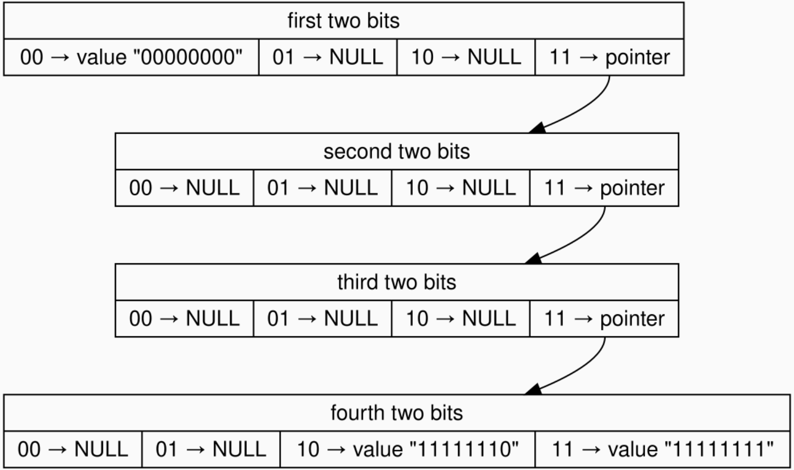
Method: a “Hash Array Mapped Trie”

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

`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

Tomorrow

First hour: excersises

Then: implementation notes and semantics