Introduction to STM

Part III: Operational Observations & Implementation Intricacies

Laurens Duijvesteijn

June 2018

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

Preface

Recap (1/3)

STM = Optimistic concurrency control

Recap (1/3)

STM = Optimistic concurrency control

A `TVar` is a transactional variable

Recap (1/3)

STM = Optimistic concurrency control

A `TVar` is a transactional variable

Operations on `TVar`s are called STM transactions

Recap (1/3)

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 (2/3)

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

Recap (2/3)

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

You can also call `retry` to rerun yourself

Recap (2/3)

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 (2/3)

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

Recap (3/3)

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

Recap (3/3)

```
TMVar a = TVar (Maybe a)

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

Recap (3/3)

`TMVar a = TVar (Maybe a)`

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

Want bounds? Add counts for read and write capacity!

Recap (3/3)

`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

Today (by request)

Operational semantics of STM

Implementation notes

Again: This is all work from other people (Composable Memory Transactions, 2006)

Operational Semantics

Can we be precice about what code is doing?

Code is all well and good, words are nice too

But sometimes we need to specify things precicely

Can we be precice about what code is doing?

Code is all well and good, words are nice too

But sometimes we need to specify things precicely

Math and logic can help communicate clearly what’s going on

A model can help us reason about what the code is doing

Where do models come from?

The authors create them! They come up with a language to communicate/specify what the code is doing.

It is a communication tool.

What does a model mean for me?

In the end, it is just that: a model

The implementation can differ from the model.

What does a model mean for me?

In the end, it is just that: a model

The implementation can differ from the model.

Or: the model doesn't cover everything in the implementation

What does a model mean for me?

In the end, it is just that: a model

The implementation can differ from the model.

Or: the model doesn't cover everything in the implementation

Some people try very hard to prove an implementation corresponds to a model

Confession time!

For me, this is the first time I looked into these things

The math does look scary; I’m not sure I got everything right

Confession time!

For me, this is the first time I looked into these things

The math does look scary; I’m not sure I got everything right

This is a “best effort” attempt; hopefully it is still interesting and mostly correct. Otherwise, I’ll stand corrected

You might have a question, though

You might have a question, though

Why do you even talk about this, if you’re not an expert?

You might have a question, though

Why do you even talk about this, if you're not an expert?

Precicely because I'm not an expert.

You might have a question, though

Why do you even talk about this, if you're not an expert?

Precicely because I'm not an expert.

This stuff can be intimidating (it was to me!)

If this helps show you that **you too** can take a naive initial step at understanding intimidating stuff; that's a win :)

With that long disclaimer..

And apologies to the original authors;

With that long disclaimer..

And apologies to the original authors; other people upon whose toes I'm stepping;

With that long disclaimer..

And apologies to the original authors; other people upon whose toes I'm stepping; and possibly this audience;

With that long disclaimer..

And apologies to the original authors; other people upon whose toes I’m stepping; and possibly this audience; let’s get to it.

Different kinds of semantics

There's multiple ways of specifying programming languages

The one we're looking at here is called "small-step operational semantics"

Different kinds of semantics

There's multiple ways of specifying programming languages

The one we're looking at here is called "small-step operational semantics"

It looks like this...

$$\boxed{\text{I/O transitions} \qquad P;\Theta \xrightarrow{a} Q;\Theta'}$$

$$\begin{array}{rcll}
\mathbb{P}[\texttt{putChar}\ c];\ \Theta & \xrightarrow{!c} & \mathbb{P}[\texttt{return ()}];\ \Theta & (PUTC) \\
\mathbb{P}[\texttt{getChar}];\ \Theta & \xrightarrow{?c} & \mathbb{P}[\texttt{return}\ c];\ \Theta & (GETC) \\
\mathbb{P}[\texttt{forkIO}\ M];\ \Theta & \rightarrow & (\mathbb{P}[\texttt{return}\ t] \mid M_t);\ \Theta \quad t \notin \mathbb{P}, \Theta, M & (FORK)
\end{array}$$

$$\frac{M \ \rightarrow\ N}{\mathbb{P}[M];\ \Theta \ \rightarrow\ \mathbb{P}[N];\ \Theta}\ (ADMIN)$$

$$\frac{M;\ \Theta, \{\} \ \stackrel{\star}{\Rightarrow}\ \texttt{return}\ N;\ \Theta', \Delta'}{\mathbb{P}[\texttt{atomic}\ M];\ \Theta \ \rightarrow\ \mathbb{P}[\texttt{return}\ N];\ \Theta'}\ (ARET) \qquad \frac{M;\ \Theta, \{\} \ \stackrel{\star}{\Rightarrow}\ \texttt{throw}\ N;\ \Theta', \Delta'}{\mathbb{P}[\texttt{atomic}\ M];\ \Theta \ \rightarrow\ \mathbb{P}[\texttt{throw}\ N];\ \Theta \cup \Delta'}\ (ATHROW)$$

$$\boxed{\text{Administrative transitions} \qquad M \ \rightarrow\ N}$$

$$M \quad \rightarrow \quad V \quad \text{if } \mathcal{V}[\![M]\!] = V \text{ and } M \not\equiv V \quad (EVAL)$$

$$\begin{array}{rcll}
\texttt{return}\ N \ \texttt{>>=}\ M & \rightarrow & M\ N & (BIND) \\
\texttt{throw}\ N \ \texttt{>>=}\ M & \rightarrow & \texttt{throw}\ N & (THROW) \\
\texttt{retry} \ \texttt{>>=}\ M & \rightarrow & \texttt{retry} & (RETRY)
\end{array} \qquad
\begin{array}{rcll}
\texttt{catch}\ (\texttt{return}\ M)\ N & \rightarrow & \texttt{return}\ M & (CATCH1) \\
\texttt{catch}\ (\texttt{throw}\ M)\ N & \rightarrow & N\ M & (CATCH2) \\
\texttt{catch retry}\ N & \rightarrow & \texttt{retry} & (CATCH3)
\end{array}$$

$$\boxed{\text{STM transitions} \qquad M;\Theta,\Delta \ \Rightarrow\ N;\Theta',\Delta'}$$

$$\begin{array}{rclll}
\mathbb{E}[\texttt{readTVar}\ r];\ \Theta, \Delta & \Rightarrow & \mathbb{E}[\texttt{return}\ \Theta(r)];\ \Theta, \Delta & \text{if } r \in dom(\Theta) & (READ) \\
\mathbb{E}[\texttt{writeTVar}\ r\ M];\ \Theta, \Delta & \Rightarrow & \mathbb{E}[\texttt{return ()}];\ \Theta[r \mapsto M], \Delta & \text{if } r \in dom(\Theta) & (WRITE) \\
\mathbb{E}[\texttt{newTVar}\ M];\ \Theta, \Delta & \Rightarrow & \mathbb{E}[\texttt{return}\ r];\ \Theta[r \mapsto M], \Delta[r \mapsto M] & r \notin dom(\Theta) & (NEW)
\end{array}$$

$$\frac{M \ \rightarrow\ N}{\mathbb{E}[M];\ \Theta, \Delta \ \Rightarrow\ \mathbb{E}[N];\ \Theta, \Delta}\ (AADMIN) \qquad \frac{M_1;\ \Theta, \Delta \ \stackrel{\star}{\Rightarrow}\ \texttt{return}\ N;\ \Theta', \Delta'}{\mathbb{E}[M_1\ \texttt{`orElse`}\ M_2];\ \Theta, \Delta \ \Rightarrow\ \mathbb{E}[\texttt{return}\ N];\ \Theta', \Delta'}\ (OR1)$$

$$\frac{M_1;\ \Theta, \Delta \ \stackrel{\star}{\Rightarrow}\ \texttt{throw}\ N;\ \Theta', \Delta'}{\mathbb{E}[M_1\ \texttt{`orElse`}\ M_2];\ \Theta, \Delta \ \Rightarrow\ \mathbb{E}[\texttt{throw}\ N];\ \Theta', \Delta'}\ (OR2) \qquad \frac{M_1;\ \Theta, \Delta \ \stackrel{\star}{\Rightarrow}\ \texttt{retry};\ \Theta', \Delta'}{\mathbb{E}[M_1\ \texttt{`orElse`}\ M_2];\ \Theta, \Delta \ \Rightarrow\ \mathbb{E}[M_2];\ \Theta, \Delta}\ (OR3)$$

Figure 4: Operational semantics of STM Haskell

First off..

This is all about single threaded Haskell (the 2006 paper)

First off..

This is all about single threaded Haskell (the 2006 paper)

The runtime: repeatedly picks a thread, then executes an IO transition. It ends up in a new state.

So: a transition is a way to move from one state to another

First off..

This is all about single threaded Haskell (the 2006 paper)

The runtime: repeatedly picks a thread, then executes an IO transition. It ends up in a new state.

So: a transition is a way to move from one state to another

There are **two** types of transitions. $\Rightarrow$ is for STM; $\rightarrow$ is for IO.

We also saw some stuff with bars

These are called rules of inference.

We also saw some stuff with bars

These are called rules of inference. They work like this:

$$\frac{\texttt{premises}}{\texttt{conclusion}}$$

We also saw some stuff with bars

These are called rules of inference. They work like this:

$$\frac{\texttt{premises}}{\texttt{conclusion}}$$

In logic, you can have multiple premises. Today, we only look at rules with one.

$$\frac{M; \Theta, \{\} \Rightarrow^{*} \texttt{pure}\ \ N; \Theta', \Delta'}{\mathbb{P}[\texttt{atomic}\ \ M]; \Theta \rightarrow \mathbb{P}[\texttt{pure}\ \ N]; \Theta'}\ (\texttt{ret})$$

$$\frac{M; \Theta, \{\} \Rightarrow^* \texttt{throw}\ \ N; \Theta', \Delta'}{\mathbb{P}[\texttt{atomic}\ \ M]; \Theta \to \mathbb{P}[\texttt{throw}\ \ N]; \Theta \cup \Delta'}$$

$$(\texttt{throw})$$

So far, we've learned/specified

The heap can only be modified by an `atomic` block (from STM)

So far, we've learned/specified

The heap can only be modified by an `atomic` block (from STM)

Exceptions propagate from STM, and leave allocation effects (new `TVar`s, as we'll see soon) but no other modifications.

Next up is reading/writing/creating `TVar`s

$$\mathbb{E}[\texttt{readTVar}\ \ r];\Theta,\Delta \Rightarrow \mathbb{E}[\texttt{pure}\ \ \Theta(r)];\Theta,\Delta$$

$$\texttt{if}\ \ r \in dom(\Theta)$$

$$(\texttt{read})$$

$$\mathbb{E}[\mathtt{writeTVar}\ \ r\ \ M]; \Theta, \Delta \Rightarrow \mathbb{E}[\mathtt{pure}\ \ (\,)]; \Theta[r \mapsto M], \Delta$$

$$\mathtt{if}\ \ r \in dom(\Theta)$$

$$(\mathtt{write})$$

$$\mathbb{E}[\texttt{newTVar}\ \ M]; \Theta, \Delta \Rightarrow \mathbb{E}[\texttt{pure}\ \ r]; \Theta[r \mapsto M], \Delta[r \mapsto M]$$

$$\texttt{if}\ \ r \notin dom(\Theta)$$

$$(\texttt{new})$$

Another pitstop

We’ve seen creation, reading and writing

Also: were those allocation effects came from

Next stop is `orElse`, which is more involved

`orElse` has three cases

1. The first transaction succeeds
2. The first transaction throws
3. The first transaction retries

$$\frac{M_1; \Theta, \Delta \Rightarrow^* \texttt{pure}\ \ N; \Theta', \Delta'}{\mathbb{E}[M_1\ \ \texttt{`orElse`}\ \ M_2]; \Theta, \Delta \Rightarrow \mathbb{E}[\texttt{pure}\ \ N]; \Theta', \Delta'}\ (\texttt{or1})$$

$$\frac{M_1; \Theta, \Delta \Rightarrow^* \texttt{throw}\ \ N; \Theta', \Delta'}{\mathbb{E}[M_1\ \ \texttt{`orElse`}\ \ M_2]; \Theta, \Delta \Rightarrow \mathbb{E}[\texttt{throw}\ \ N]; \Theta', \Delta'}\ (\texttt{or2})$$

Hold on right there!

Why is there a Θ' there, and not a Θ?

Hold on right there!

Why is there a Θ' there, and not a Θ?

This means that the heap modifications are still there. Is that a problem?

Hold on right there!

Why is there a Θ' there, and not a Θ?

This means that the heap modifications are still there. Is that a problem?

Remember the rule from before (in `IO` instead of `STM`)?

Hold on right there!

Why is there a Θ' there, and not a Θ?

This means that the heap modifications are still there. Is that a problem?

Remember the rule from before (in `IO` instead of `STM`)?That addresses this.

$$\frac{M_1; \Theta, \Delta \Rightarrow^* \texttt{retry}; \Theta', \Delta'}{\mathbb{E}[M_1 \ \texttt{`orElse`} \ M_2]; \Theta, \Delta \Rightarrow \mathbb{E}[M_2]; \Theta, \Delta} \ (\texttt{or3})$$

There's something missing, though

There's no explicit rule for what happens when a `retry` is reached, while in `IO`.

There's something missing, though

There's no explicit rule for what happens when a `retry` is reached, while in `IO`.

Because the system cannot make any progress when this happens. It blocks.

There’s something missing, though

There’s no explicit rule for what happens when a `retry` is reached, while in `IO`.

Because the system cannot make any progress when this happens. It blocks.

Other threads can make progress, and if any `TVar` of the transaction changes, the thread can proceed.

Implementation

So I mentioned this log a couple of times..

Its purpose is to find conflicts before a transaction commits

Let's see how it works

Each thread has a log

It gets created as soon as a transaction starts (e.g. the `atomic` block is encountered)

Each log contains:

1. A parent pointer (may be null)
2. A number of log entries

Log entries

One exists for each `TVar` that's touched by the transaction.

These have three things:

1. The `TVar`
2. Its value when the transaction started
3. Its current value

How do we know there's conflicts?

For a transaction, we check all entries in the log.

There's a conflict if one of the `TVar`s changed during the transaction.

How do we know there's conflicts?

For a transaction, we check all entries in the log.

There's a conflict if one of the `TVar`s changed during the transaction.

(Checking this is linear in the amount of `TVar`s accessed)

Aborting transactions

Aborting a transaction simply means discarding its log

Aborting transactions

Aborting a transaction simply means discarding its log

Writes happen to the log, in the “new value once transaction commits” slot, so they don’t have any effects.

Aborting transactions

Aborting a transaction simply means discarding its log

Writes happen to the log, in the “new value once transaction commits” slot, so they don’t have any effects.

Reads also happen from the log, so a transaction can see the values it has written. This also helps with A-B-A problems.

Committing transactions

Loop over the transaction log and, for each entry, write the new value to the `TVar`.

In the single threaded runtime, this is enough. Only one thread can commit at the same time.

Multicore would need some sort of synchronization: or you'll have trouble when two threads commit to the same `TVar`s in different orders

Nontermination guarantees

There's cases where transactions can loop indefinitely because of the values they read.

Nontermination guarantees

There's cases where transactions can loop indefinitely because of the values they read.

It's obvious after you see it, but I never thought of it.

Nontermination guarantees

There's cases where transactions can loop indefinitely because of the values they read.

It's obvious after you see it, but I never thought of it.

Fix: before switching to a thread in a transaction, run the check routine. (remember: single threaded runtime)

Avoiding busy wait

We mentioned: there's no busy wait because transactions are only reran when their inputs change.

Avoiding busy wait

We mentioned: there's no busy wait because transactions are only reran when their inputs change.

But: how does the runtime know which threads to rerun?

Avoiding busy wait

We mentioned: there's no busy wait because transactions are only reran when their inputs change.

But: how does the runtime know which threads to rerun?

Each `TVar` has a list of blocking transactions! When a thread commits to a `TVar`, it wakes up another.

`orElse` and transaction nesting

The presence of `orElse` means we have to have some form of transaction nesting

`orElse` and transaction nesting

The presence of `orElse` means we have to have some form of transaction nesting (to isolate excecution of alternatives).

`orElse` and transaction nesting

The presence of `orElse` means we have to have some form of transaction nesting (to isolate excecution of alternatives).

For this, we create a nested log. (Remember the parent pointer?)

`orElse` and transaction nesting

The presence of `orElse` means we have to have some form of transaction nesting (to isolate excecution of alternatives).

For this, we create a nested log. (Remember the parent pointer?)

If a nested transaction finishes, we validate its log and that of the parent. Both good? Then merge the two logs.

In case of `retry `orElse` retry`

What `TVar`s should we wait on?

In case of `retry `orElse` retry`

What `TVar`s should we wait on? A: The union of those accessed in both transactions!

In case of `retry `orElse` retry`

What `TVar`s should we wait on? A: The union of those accessed in both transactions!

This is so all the `TVar`s which have been accessed can get the current thread ID on their list of waiting threads.

In case of `retry `orElse` retry`

What `TVar`s should we wait on? A: The union of those accessed in both transactions!

This is so all the `TVar`s which have been accessed can get the current thread ID on their list of waiting threads.

Observe: this log is only used for installing these waiting entries, or its discarded.

Closing

Closing

We’ve seen: Operational semantics of STM, and some implementation notes of the single threaded version.

Closing

We've seen: Operational semantics of STM, and some implementation notes of the single threaded version.

That's not how the real thing works, but it is a good didactic model IMO.

Closing

Hopefully this has been interesting to you!

The slides are all going to be online: `duijf/stm-course`

Closing

Hopefully this has been interesting to you!

The slides are all going to be online: `duijf/stm-course`

Feedback: feel free to tell me or write on a GitHub ticket!

Fin.