Alternative Software Transactional Memory Implementation in Haskell

Master Thesis

Lasse Folger

05.04.2017

Motivation

```
type Account = MVar Int

transfer :: Account -> Account -> Int -> IO ()
transfer src dst am = do
  a1 <- takeMVar src
  a2 <- takeMVar dst
  putMVar src (a1 - am)
  putMVar dst (a2 + am)
```

MVar

Thread 1:

```
transfer acc1 acc2 50
```

Thread 2:

```
transfer acc2 acc1 50
```

MVar

Thread 1:

```
a1 <- takeMVar acc1
a2 <- takeMVar acc2
writeMVar acc1 (a1 - 50)
writeMVar acc2 (a2 + 50)
```

Thread 2:

```
b1 <- takeMVar acc2
b2 <- takeMVar acc1
writeMVar acc2 (b1 - 50)
writeMVar acc1 (b2 + 50)
```

MVar

Thread 1:

```
a1 <- takeMVar acc1
a2 <- takeMVar acc2
writeMVar acc1 (a1 - 50)
writeMVar acc2 (a2 + 50)
```

Thread 2:

```
b1 <- takeMVar acc2
b2 <- takeMVar acc1
writeMVar acc2 (b1 - 50)
writeMVar acc1 (b2 + 50)
```

$\Rightarrow$ Deadlock

Use Transactions

```
type Account = TVar Int

transfer :: Account -> Account -> Int -> STM ()
transfer src dst am = do
  a1 <- readTVar src
  a2 <- readTVar dst
  writeTVar src (a1 - am)
  writeTVar dst (a2 + am)
```

TVar

Thread 1:

```
atomically $
  transfer acc1 acc2 50
```

Thread 2:

```
atomically $
  transfer acc2 acc1 50
```

TVar

Thread 1:

```
atomically $
  transfer acc1 acc2 50
```

Thread 2:

```
atomically $
  transfer acc2 acc1 50
```

$\Rightarrow$ works fine, because transactions provide ACI(D) properties

Current Implementation (Control.Concurrent.STM)

- *writeTVar*, *readTVar* and *newTVar* modify TVars
- *retry* and *orElse* alter the control flow
- *atomically* executes a transaction
- composition via bind operator (or do)

Transactional Log

- one log per transaction
- necessary for consistency
- three elements per log entry
  - **TVar**
  - **expectedValue**
  - **currentValue**

Modify Operations

- **newTVar**: creates a new, initialized TVar
- **writeTVar**: updates *currentValue* in log entry
- **readTVar**: reads TVar from log or actual TVar

Example

```
transaction = do
  a <- readTVar t1
  writeTVar t2 a
  readTVar t2
```

Example

```
transaction = do
  a <- readTVar t1
  writeTVar t2 a
  readTVar t2
```

Log after first action:

```
[(t1,a,a)]
```

Example

```
transaction = do
  a <- readTVar t1
  writeTVar t2 a
  readTVar t2
```

Log after second action:

```
[(t1,a,a), (t2,b,a)]
```

Example

```
transaction = do
  a <- readTVar t1
  writeTVar t2 a
  readTVar t2
```

Log after thrid action:

```
[(t1,a,a), (t2,b,a)]
```

atomically :: STM a -> IO a

1. compute the log
2. lock TVars
3. validate the log
4. if valid then commit
5. else roll back

Validation

1. compare *expectedValue* to the value in the actual TVar
2. if all values match return valid
3. else return invalid

Problem

Thread 1:

```
a1 <- readTVar acc1
a2 <- readTVar acc2
writeTVar acc1 (a1 - 50)
writeTVar acc2 (a2 + 50)
```

Thread 2:

```
b1 <- readTVar acc2
b2 <- readTVar acc1
writeTVar acc2 (b1 - 50)
writeTVar acc1 (b2 + 50)
```

Log:

```
[(acc1, a1, a1-50),
 (acc2, a2, a2+50)]
```

Log:

```
[(acc2, b1, b1-50),
 (acc1, b2, b2+50)]
```

Problem

Thread 1:

```
[(acc1, a1, a1-50),
 (acc2, a2, a2+50)]
```

Thread 2:

```
[(acc2, b1, b1-50),
 (acc1, b2, b2+50)]
```

Problem

Thread 1:

```
[(acc1, a1, a1-50),
 (acc2, a2, a2+50)]
```

Thread 2:

```
[(acc2, b1, b1-50),
 (acc1, b2, b2+50)]
```

$\Rightarrow$ either sequential or one transaction is rolled back

Critical TVar

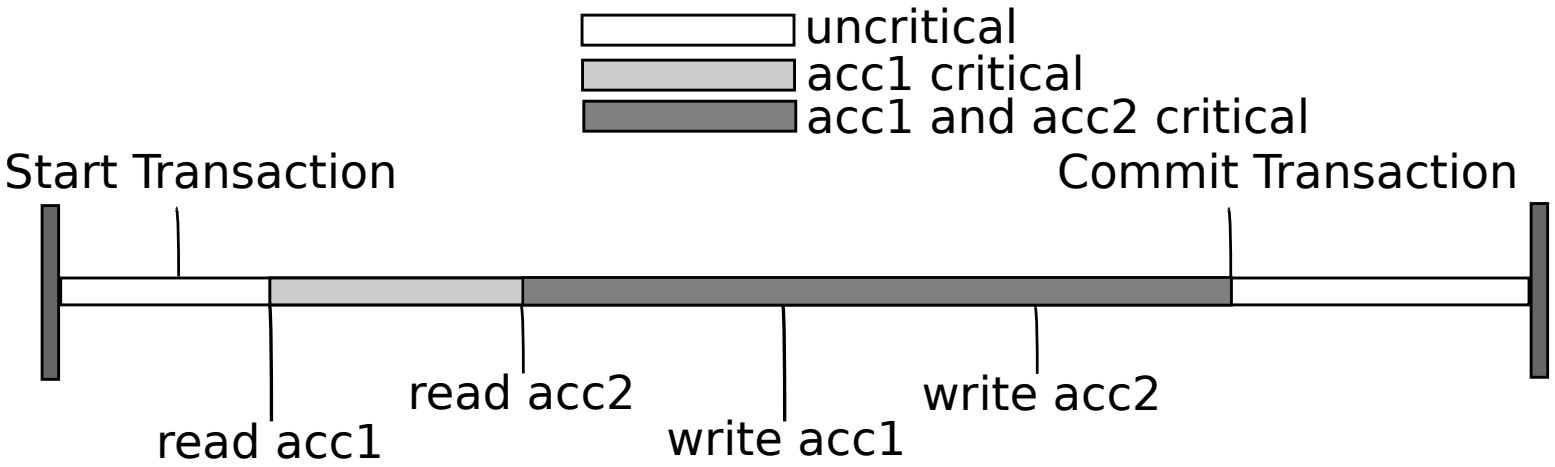
- critical between read and commit
- modifications to critical TVars cause rollback

Critical TVar

- critical between read and commit
- modifications to critical TVars cause rollback

$\Rightarrow$ minimize the time TVars are critical

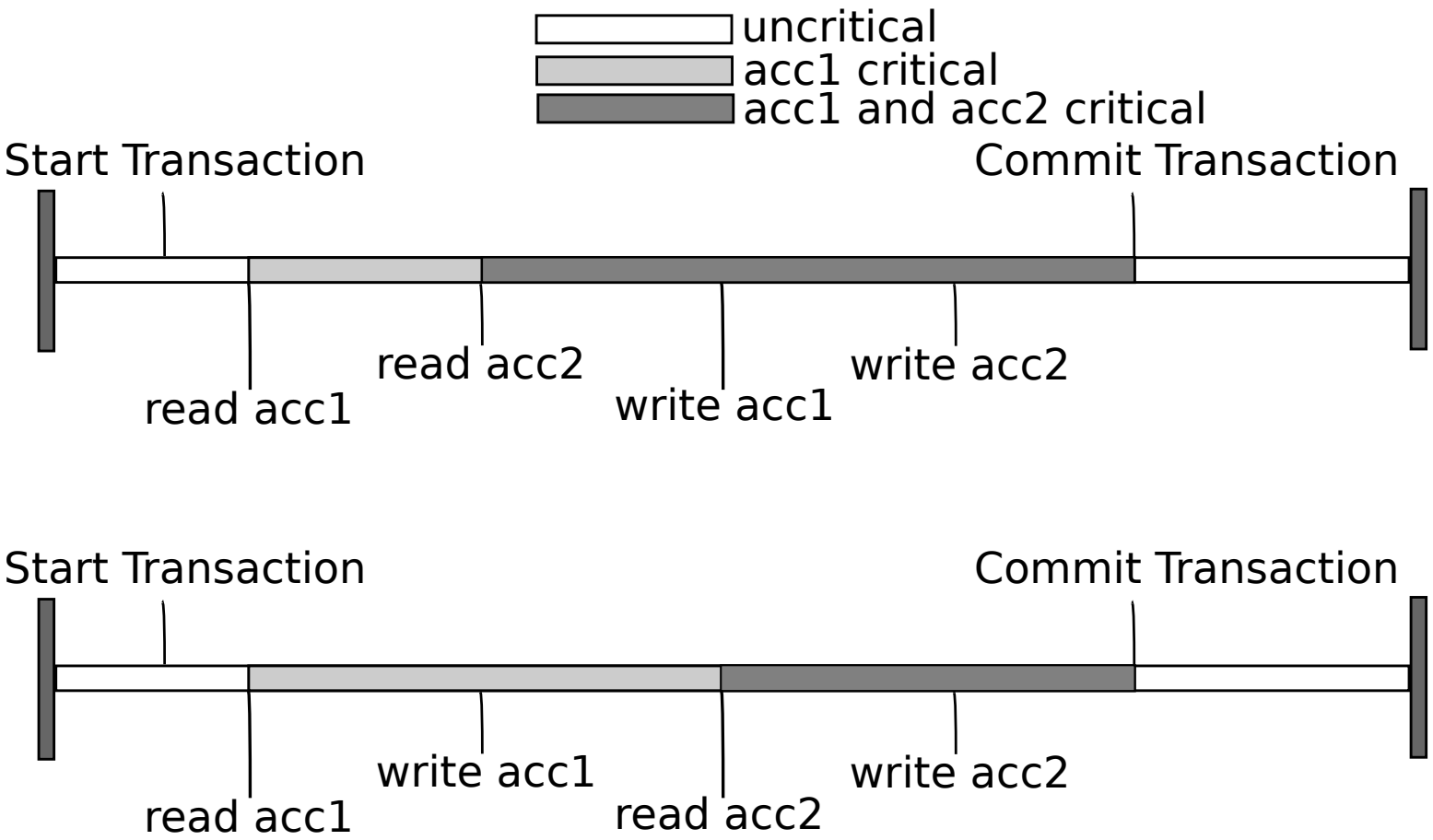
Critical TVar



Idea

```
transfer = do
  a1 <- readTVar acc2
  writeTVar acc2 (a1 - 50)
  a2 <- readTVar acc1
  writeTVar acc1 (a2 + 50)
```

Critical TVar



Idea

- delay the evaluation of **readTVar** to commit phase
- no TVars is critical at all
- **writeTVar** does not need the value in the computation phase

Idea does not work

```
limitedTransfer src dst am = do
    a1 <- readTVar src
    if a1 < am
      then return ()
      else do a2 <- readTVar dst
              writeTVar src (a1 - am)
              writeTVar dst (a2 + am)
```

$\Rightarrow$ idea does not work because the value is needed.

Solution

- delay evaluation as far as possible
- evaluate reads just before they are needed..
- ..or in the commit phase

When is a value needed?

- branch conditions
  - if-then-else
  - case
  - patternmatching
  - guards
- IO-actions $\Rightarrow$ not allowed in STM

Example

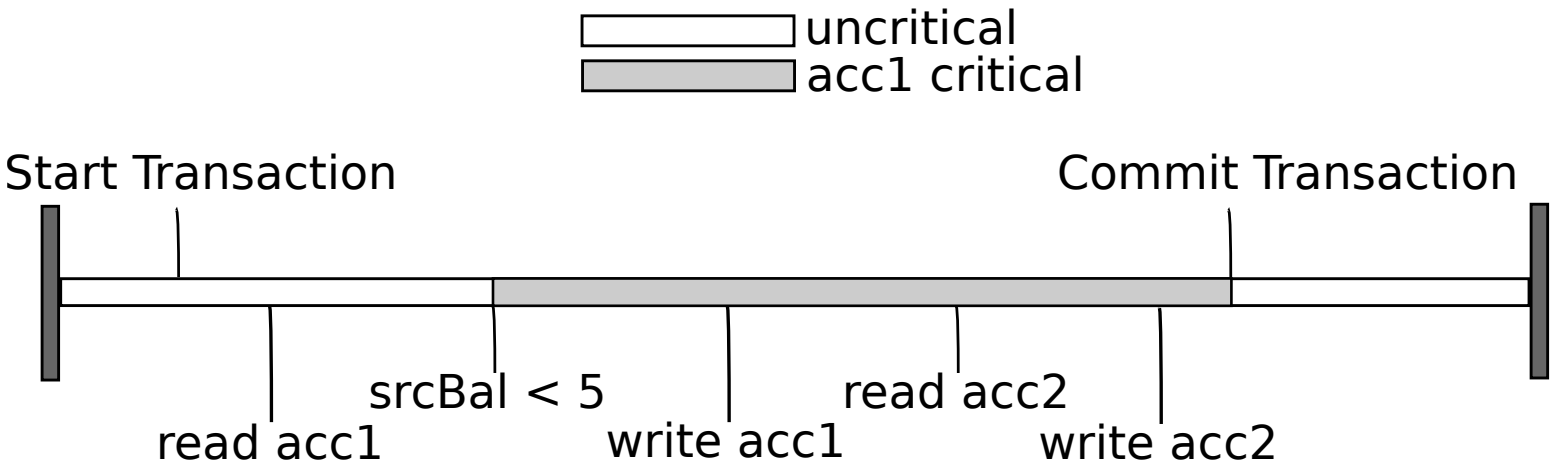
```
type Account = TVar Int

transfer :: Account -> Account -> Int -> STM ()
transfer src dst am = do
  a1 <- readTVar src
  a2 <- readTVar dst
  writeTVar src (a1 - am)
  writeTVar dst (a2 + am)
```

Example: limitedTransfer acc1 acc2 5

```
a1 <- readTVar acc1
if a1 < 5
  then return ()
  else do a2 <- readTVar acc2
          writeTVar acc1 (a1 - am)
          writeTVar acc2 (a2 + am)
```

Example: limitedTransfer acc1 acc2 5

```
a1 <- readTVar acc1
if a1 < 5
  then return ()
  else do a2 <- readTVar acc2
          writeTVar acc1 (a1 - am)
          writeTVar acc2 (a2 + am)
```



Implementation

- pure Haskell implementation
- state monad
- computation phase modifies the state
- commit phase uses state to validate and commit

State

- read log
- write log
- unevaluated reads

Delayed Evaluation

- search TVar in write log
- **readTVar** creates an read expression and extends write log
- **unsafePerformIO** :: IO a $\rightarrow$ a
- IO action logs the information in read log

Computation Phase

- **writeTVar** modifies write log
- **newTVar** does not access the state
- computation phase is evaluation of the STM action

Commit Phase

1. lock TVars in write log
2. validate read log
3. evaluate remaining reads
4. publish modifications

Evaluation

- STM is universal tool
- unlimited possibilities
- testing is challenging

Evaluation

- tested specific cases
- increase in level of concurrency
- results meet the expectations

	GHC	Project	STMLA
medium concurrency	3.0420	3.2830	2.9225
low concurrency	3.0670	3.3110	2.9425
high concurrency	3.1520	3.4335	2.9455

Future Work: True Consistency

```
transaction = do
  a <- readTVar t1
  b <- readTVar t2
  if b == a
    then return ()
    else loop
```

Future Work: Exception Handling

- exception handling
- validate before passing an exception
- if invalid roll back
- unlock TVars when expection occurs

Future Work

- invariants
- C library with compiler integration
- further performance testing

Questions about...

- ...Control.Concurrent.STM?
- ...rollback avoidance?
- ...my implementation?
- ...something else?