A Haskell-Implementation of STM Haskell with Early Conflict Detection

David Sabel

Goethe-University, Frankfurt, Germany

ATPS 2014, Kiel

Software Transactional Memory (STM)

- treats **shared memory** operations as **transactions**
- provides **lock-free** and **very convenient** concurrent programming

STM Haskell

- STM library for Haskell
- introduced by Harris et.al, PPoPP'05

Our contribution: an alternative implementation of STM Haskell

- **earlier** conflict detection
- based on a **correct** concurrent program calculus for STM Haskell, Schmidt-Schauß & S., ICFP'13

Transactional Variables:

```
TVar a
```

Primitives to form STM-transactions `STM a`:

```
newTVar     :: a -> STM (TVar a)
readTVar    :: TVar a -> STM a
writeTVar   :: TVar a -> a -> STM ()

return      :: a -> STM a
(>>=)       :: STM a -> (a -> STM b) -> STM b

retry       :: STM ()
orElse      :: STM a -> STM a -> STM a
```

Executing an STM-transaction:

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

Semantics: the transaction-execution is

- **atomic**: all or nothing, effects are indivisible, and
- **isolated**: concurrent evaluation is not observable

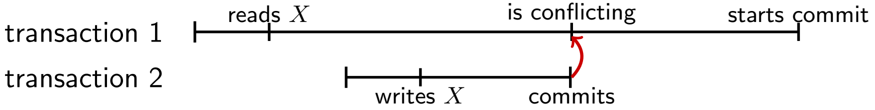
- GHC STM (**Harris et.al., PPoPP'05**): implementation in the Glasgow Haskell Compiler: implemented **in C**, deeply embedded in the runtime system
- **Huch & Kupke, IFL'05**: Implementation in **Haskell 98**
- **Du Bois, SC'11**: STM implementation in Haskell based on the **TL 2 algorithm** by **Dice et.al., DISC'06**
- SHFSTM: Implementation in GHC Haskell, **early conflict detection**, based on a program calculus CSHF, **correctness** proved in **Schmidt-Schauß & S., ICFP'13**,

Common Features of all Implementations

- transactions perform reads and writes on **local working copies**
- **commit phase**: local content is copied to global `TVars`
- transactions use a **transaction log** for book-keeping of operations
- **conflict** if the global content of already read TVar changes

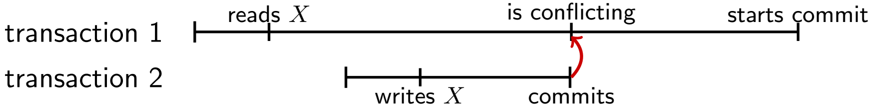
GHC STM; Huch & Kupke:

- transaction log: for every TVar the **old** and the **new** value
- transaction detects conflict **by itself** (inspecting its transaction log): (old value $\neq$ current value) $\implies$ conflict
- moment of conflict detection: **commit phase** (and temporary, GHC)



GHC STM; Huch & Kupke:

- transaction log: for every TVar the **old** and the **new** value
- transaction detects conflict **by itself** (inspecting its transaction log): (old value $\neq$ current value) $\implies$ conflict
- moment of conflict detection: **commit phase** (and temporary, GHC)



SHFSTM:

- transaction log: the new value for every TVar, information which TVars were read, written, ...
- every TVar has a **notification list** of thread identifiers
- committing thread **notifies** conflicting threads in the notification lists of written TVars
- moment of conflict detection: when conflict occurs, i.e. **early**

Nontermination and Early Conflict Detection

```
trans1 tvar = atomically $
 do c <- readTVar tvar
    if c then
      let loop = loop in loop
     else return ()
```

```
trans2 tvar =
 atomically (writeTVar tvar False)
```

- Specification: atomic execution of `trans1`: all or nothing!
  $\Longrightarrow$ execution of both transactions always terminates.

Nontermination and Early Conflict Detection

```
trans1 tvar = atomically $
 do c <- readTVar tvar
    if c then
      let loop = loop in loop
     else return ()
```

```
trans2 tvar =
 atomically (writeTVar tvar False)
```

- Specification: atomic execution of `trans1`: all or nothing!
  $\Longrightarrow$ execution of both transactions always terminates.
- GHC STM: temporary check of transaction log detects conflict, but program sometimes fails, due to loop detection.
- Huch & Kupke'05; du Bois'11: nontermination, no temporary conflict detection
- SHFSTM: termination, commit phase of `trans2` notifies `trans1`

Implementation of SHFSTM: TVars

```
newtype TVarA a = TVarA (MVar (ITVar a))

data ITVar a = TV
   { globalContent :: MVar a
   , localContent  :: MVar (Map ThreadId (IORef [a]))
   , notifyList    :: MVar (Set ThreadId)
   , lock          :: MVar ThreadId
   , waitingQueue  :: MVar [MVar ()] }
```

- all parts are mutable and protected by MVars
- local copies for all threads are stored in `localContent`
- `notifyList` holds thread identifiers of conflicting transactions
- `lock` is used during commit
- `waitingQueue` is used to block other threads if TVar is locked

Implementation: Transaction Log

```
data Log = Log {
  readTVars   :: Read,
  tripleStack :: [(Accessed,Written,Created)],
  lockingSet  :: Locked }
```

- *Read*, *Accessed*, *Written*, *Created*, and *Locked* are **heterogenous** sets of TVars
- All operations on the sets do not depend on the content type $\Rightarrow$ **existential types** can be used

Implementation: Transaction Log

```
data Log  =  Log {
  readTVars   :: Set TVarAny,
  tripleStack :: [(Set TVarAny,Set TVarAny,Set TVarAny)],
  lockingSet  :: Set TVarAny }
```

- *Read*, *Accessed*, *Written*, *Created*, and *Locked* are **heterogenous** sets of TVars
- All operations on the sets do not depend on the content type $\Rightarrow$ **existential types** can be used

```
data    TVarAny = forall a. TVarAny (TVarId, MVar (ITVar a))
```

- A TVar is a pair of `TVarA a` and `TVarAny`:

```
newtype TVar a  = TVar (TVarA a,TVarAny)
newtype TVarA a = TVarA (MVar (ITVar a))
```

Implementation: Transaction Log

```
data Log  =  Log {
  readTVars   :: Set TVarAny,
  tripleStack :: [(Set TVarAny,Set TVarAny,Set TVarAny)],
  lockingSet  :: Set TVarAny }
```

- *Read*, *Accessed*, *Written*, *Created*, and *Locked* are **heterogenous** sets of TVars
- All operations on the sets do not depend on the content type $\Rightarrow$ **existential types** can be used

```
data    TVarAny = forall a. TVarAny (TVarId, MVar (ITVar a))
```

- A TVar is a pair of `TVarA a` and `TVarAny`:

```
newtype TVar a  = TVar (TVarA a,TVarAny)
newtype TVarA a = TVarA (MVar (ITVar a))
```

- Invariant:
  both `MVar (ITVar a)`-components always point to the same object

Implementation: STM-Transactions

- Like an embedded language:

```
data STM a = Return a
           | Retry
           | forall b. NewTVar b (TVar b -> STM a)
           | forall b. ReadTVar  (TVar b) (b -> STM a)
           | forall b. WriteTVar (TVar b) b (STM a)
           | forall b. OrElse (STM b) (STM b) (b -> STM a)
```

- additional argument stores the continuation
- existential types to hide intermediate types

```
readTVar :: TVar a -> STM a
readTVar a = ReadTVar a return
...
instance Monad STM where
 return  = Return
 m >>= f = case m of
             ReadTVar x cont -> ReadTVar x (cont >>= f)
             ...
```

Implementation: atomically

- atomically executes the embedded language

```
atomically :: STM a -> IO a
atomically act = do
 tlog <- emptyTLOG
 catch (performSTM tlog act)
       (\e -> case e of RetryException ->
                          do uninterruptibleMask_ (globalRetry tlog)
                             atomically act)
```

- Exceptions are used to notify the conflicting thread

```
notify :: [ThreadId] -> IO ()
notify []        = return ()
notify (tid:xs) = throwTo tid RetryException >> notify xs
```

- performSTM calls specific functions for every operation

```
performSTM tlog (Return a)        = commit tlog >> return a
performSTM tlog Retry             = waitForExternalRetry
performSTM tlog (ReadTVar x cont) = do c <- readTVarWithLog tlog x
                                       performSTM tlog (cont c)
```

Commit Phase

```
commit :: TLOG  -> IO ()
commit tlog = do
 writeStartWithLog tlog -- lock the TVars
 writeClearWithLog tlog -- remove own notify entries
 sendRetryWithLog tlog  -- notify conflicting threads
 writeTVWithLog tlog    -- copy local content into global TVars
 writeTVnWithLog tlog   -- create the new TVars
 writeEndWithLog tlog   -- clear the local TVar-stacks
 unlockTVWithLog tlog   -- unlock the TVars, unblock waiting threads
```

- locking the TVars is not atomic (difference to $CSHF$)
- locks are taken in total order
- if not all locks are available, already held locks are released, since the thread maybe conflicting

Experimental Results

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

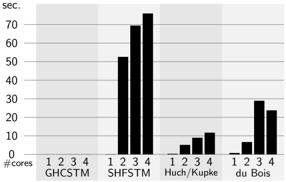
Test environment:

- Intel i7-2600 CPU (4 cores)
- compiled with GHC 7.4.2 and -O2 on Linux
- mean runtime of 15 runs
- 4 libraries: GHC STM, SHFSTM, Huch & Kupke, du Bois

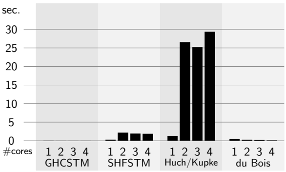
Tests:

- Some tests used of the Haskell STM Benchmark
  http://www.bscmsrc.eu/software/haskell-stm-benchmark
- Some own tests

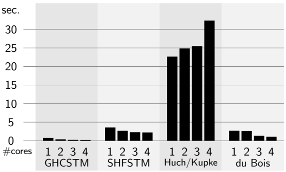
Experimental Results (2)



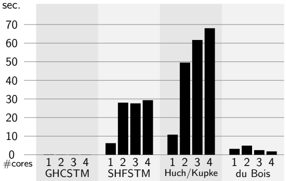
Shared Int



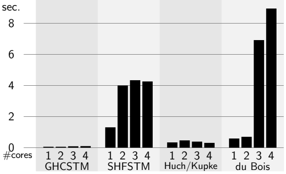
Sum of Shared TVars



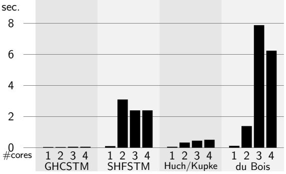
Sudoku



Linked List



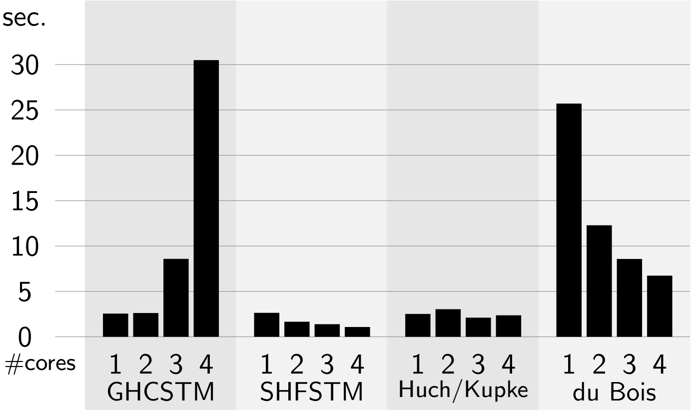
Binary Tree



Hash Table

- Shared Int: 200 threads increase the value of a single TVar
- Sum of Shared TVars: 200 threads write the sum of 200 TVars into the last TVar
- Sudoku: Parallel Sudoku-Solver, cells are stored in TVars
- Linked List: 200 threads perform 100 operations on a linked list built from TVars
- Binary Tree: 200 threads perform 100 operations on a binary tree built from TVars
- Hash Table: 100 threads perform 100 operations on a hashtable built from TVars

Experimental Results (3)



- 40 threads: every thread reads the same 5 TVars,
- for every read: compute ackermann(i,3) where i is between 6 and 8 depending on the thread number
- write the sum into the last TVar

Conclusion

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

- correct STM implementations require correct treatment of nonterminating transactions
- the SHFSTM-implementation works and detects conflicts early
- GHC STM performs in most cases much better
- implementation of SHFSTM uses exceptions as programming primitive

Further work

- optimize the implementation using concurrent data structures
- implementation in C as part of the runtime system?