

STM Haskell

Prof. Dr. David Sabel

LFE Theoretische Informatik



Letzte Änderung der Folien: 16. Januar 2020

Software Transactional Memory in Haskell

- STM in Haskell von Harris, Marlow, Peyton Jones und Herlihy 2005 eingeführt
- Implementiert in der Bibliothek `Control.Concurrent.STM`
- Erstes Papier, das `retry` und `orElse` einführt.

Übersicht

- 1 Einleitung
- 2 Beispiele
- 3 Transaktionen kombinieren
- 4 Weitere Beispiele
- 5 Implementierung

TCS | 11 STM Haskell | WS 2019/20

2/47

Einl. Beispiele T. kombinieren W. Beisp. Implement.

Design

- **Trennung** von Operationen auf Transaktionsvariablen und normalem IO
- mithilfe des **Typsystems**
- Wie IO-Monade gibt es eine **STM-Monade**
- Transaktionen sind vom Typ `STM a`
- `atomically :: STM a -> IO a` überführt eine STM-Aktion in eine IO-Operation

Semantik: `atomically t` führt die Transaktion `t` atomar aus.

- Beachte: Es gibt keine Umkehrung vom Typ `IO a -> STM a`!
- Halbwahrheit: `unsafeIOToSTM :: IO a -> STM a`
- Da STM eine Monade ist, kann die `do`-Notation und auch `>>=`, `>>` und `return` verwendet werden.

Transaktionsvariablen TVar

- `data TVar a = ...`
- `newTVar :: a -> STM (TVar a)`
Erzeugt eine neue TVar mit Inhalt
- `readTVar :: TVar a -> STM a`
Liest den momentanen Wert einer TVar
- `writeTVar :: TVar a -> a -> STM ()`
Schreibt einen neuen Wert in die TVar
- `newTVarIO :: TVar a -> a -> IO (TVar a)`
Erzeugen einer TVar in der IO Monade

retry

- `retry :: STM a`
- Bricht die Transaktion ab und startet sie neu
- **Implementierung:** Neustart erst dann, wenn sich die Werte von TVars geändert haben
- Damit wird implizit blockiert!

Transaktionen

```
atomically (  
  do  
    ... Zugriffe auf TVar's ...  
  und  
    ... pure funktionale Berechnungen  
  
  ... aber kein IO!  
  
)
```

Anwendung: Mit retry-auf Veränderung warten

Beispiel:

- Prozesse verändern einen Zähler
- Es gibt einen Prozess, der den Zähler auf dem Bildschirm anzeigt
- Updates des Zählers nicht ständig, sondern nur bei Änderungen von mindestens 1000

Anwendung: Mit retry-auf Veränderung warten (2)

Haupt-Programm:

```
main = do
  -- TVar fuer den Zaehler
  tv <- (newTVarIO 0)::IO (TVar Integer)
  -- 10 worker-Threads erzeugen
  mapM_ forkIO (replicate 10 (worker tv))
  -- aktualisieren der Anzeige starten
  updateDisplay tv
```

Anwendung: Mit retry-auf Veränderung warten (3)

Zufälliges Ändern des Zählers:

```
worker tv =
  do z <- randomRIO (-100,100)
     threadDelay 10000
     atomically $ do
       old <- readTVar tv
       writeTVar tv (z+old)
  worker tv
```

Anwendung: Mit retry-auf Veränderung warten (4)

Aktualisieren der Anzeige:

```
updateDisplay tv =
  do b <- readTVarIO tv
     printNext b -- einmal anzeigen
     loop b      -- in Schleife einsteigen
  where
    loop b = do next <- atomically $
      do c <- readTVar tv
         if abs (b-c) < 1000
         then retry
         else return c
       printNext next
       loop next
```

```
printNext i = putStr $ "\r" ++ show i ++ "      "
```

Anwendung: Mit retry-auf Veränderung warten (4)

Realeres Beispiel:

- Fenstermanager
- Zeichne Fenstern neu, wenn sich die Positionsdaten ändern
- Rendering wartet mit retry auf Positionsänderung.

Beispiele

- Wir betrachten weitere Beispiele zur Programmierung
- Semaphore, MVars, Philosophen, Kanäle

MVar innerhalb von STM implementieren

```
type MVar a = TVar (Maybe a)
```

Zur Erinnerung:

```
data Maybe a = Nothing
              | Just a
```

MVar erzeugen:

```
newEmptyMVar :: STM (MVar a)
newEmptyMVar = newTVar Nothing
```

Beispiel: Binärer Semaphore

```
type Semaphore = TVar Bool
```

```
newSem :: Bool -> IO Semaphore
newSem k = newTVarIO k -- k True/False
```

```
wait :: Semaphore -> STM ()
wait sem = do b <- readTVar sem
              if b
                then writeTVar sem False
                else retry
```

```
signal :: Semaphore -> STM ()
signal sem = writeTVar sem True
```

MVar innerhalb von STM implementieren (2)

takeMVar-Operation:

```
takeMVar :: MVar a -> STM a
takeMVar mv = do
  v <- readTVar mv
  case v of
    Nothing -> retry
    Just val -> do
      writeTVar mv Nothing
      return val
```

MVar innerhalb von STM implementieren (3)

putMVar-Operation:

```
putMVar :: MVar a -> a -> STM ()
putMVar mv val = do
  v <- readTVar mv
  case v of
    Nothing -> writeTVar mv (Just val)
    Just val -> retry
```

Transaktionen kombinieren

- `>>=`, `>>`, `do` erstellen aus Transaktionen zusammengesetzte **Sequenzen**.
- Beispiel:

```
swapMVar :: MVar a -> a -> STM a
swapMVar mv val = do
  res <- takeMVar mv
  putMVar val
  return res
```
- `swapMVar` wird **atomar** ausgeführt, da es eine Transaktion ist.

Transaktionen kombinieren: Auswahl

- `orElse :: STM a -> STM a -> STM a`
- kombiniert zwei Transaktion zu einer
- wenn die erste Transaktion erfolgreich ist, dann auch die kombinierte
- wenn die erste Transaktion ein `retry` durchführt, dann wird die zweite Transaktion durchgeführt.
- wenn beide Transaktionen ein `retry` durchführen, so wird die gesamte Transaktion neu gestartet.

Nochmal zur MVar-Implementierung

Nichtblockierendes putMVar

```
tryPutMVar :: MVar a -> a -> STM Bool
tryPutMVar mv val
= ( do
    putMVar mv val
    return True
  )
  `orElse`
  (return False)
```

orElse erweitern

Aufbauend auf dem [binären orElse](#) kann man problemlos neue Kombinatoren definieren.

```
mergeSTM :: [STM a] -> STM a
mergeSTM transactions = foldl1 orElse transactions
```

```
foldl1 op [x]      = x
foldl1 op (x:xs) = x 'op' (foldl e op xs)
```

`mergeSTM [t1,...,tn]` ergibt
(t1 'orElse' (t2 'orElse' (t3 'orElse' ... ())))

Die erste erfolgreich durchgeführte Transaktion bestimmt das Ergebnis.

Speisende Philosophen in STM (2)

```
philosoph :: Int->Buffer String->Semaphore->Semaphore->IO ()
philosoph n out fork1 fork2 = do
  atomically $ put out ("Philosoph " ++ show n ++ " denkt.")
  atomically $ do wait fork1
                  wait fork2
  atomically $ put out ("Philosoph " ++ show n ++ " isst.")
  atomically $ do signal fork1
                  signal fork2
philosoph n out fork1 fork2
```

Speisende Philosophen in STM

siehe: <http://computationalthoughts.blogspot.com/2008/03/some-examples-of-software-transactional.html>

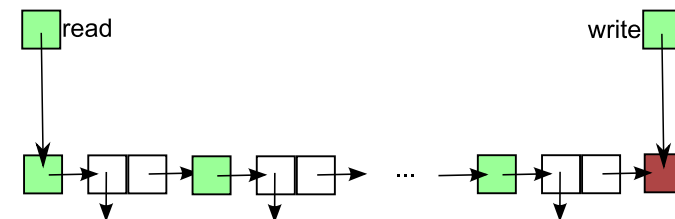
```
simulation n = do
  forks <- sequence (replicate n (newSem True))
  outputBuffer <- newBuffer
  sequence_ [forkIO (philosoph
                    i
                    outputBuffer
                    (forks!!i)
                    (forks!!((i+1)'mod'n)))
            | i <- [0..n-1]]
  output outputBuffer
```

```
output buffer = do
  str <- atomically $ get buffer
  putStrLn str
  output buffer
```

Kanäle mit STM-Haskell

- Analog zu MVars: Mit TVars zusammensetzen
- Unterscheidung zwischen leer und voll gibt es nicht bei TVars
- Deshalb Konstruktor TNil für leer

```
type TKanal a = (TVar (TVarListe a), TVar (TVarListe a))
type TVarListe a = TVar (TListe a)
data TListe a = TNil | TCons a (TVarListe a)
```



- rote Box = TVar gefüllt mit TNil
- grünen Boxen = TVars gefüllt mit TCons h t (weiße Doppelbox) oder für read und write auf Anfang/Ende

Kanäle mit STM-Haskell: Erzeugen

```
neuerTKanal :: STM (TKanal a)
neuerTKanal = do
  hole <- newTVar TNil
  read <- newTVar hole
  write <- newTVar hole
  return $ (read, write)
```

Kanäle mit STM-Haskell: Schreiben

```
schreibe :: TKanal a -> a -> STM ()
schreibe (read,write) val = do
  newEnd <- newTVar TNil
  oldEnd <- readTVar write
  writeTVar write newEnd
  writeTVar oldEnd (TCons val newEnd)
```

Kanäle mit STM-Haskell: Lesen

```
lese :: TKanal a -> STM a
lese (read,write) =
  do
    listHead <- readTVar read
    tryHead <- readTVar listHead
    case tryHead of
      TNil -> retry
      TCons val rest -> do writeTVar read rest
                          return val
```

Kanäle mit STM-Haskell: Duplizieren

```
dupliziere :: TKanal a -> STM (TKanal a)
dupliziere (read,write) =
  do
    hole <- readTVar write
    new_read <- newTVar hole
    return (new_read,write)
```

```
undoLese :: TKanal a -> a -> STM ()
undoLese (read,write) val =
  do
    listHead <- readTVar read
    newHead <- newTVar (TCons val listHead)
    writeTVar read newHead
```

Beachte: Das ging mit MVars nicht!

```
istLeer :: TKanal a -> STM Bool
istLeer (read,write) =
  do
    listHead <- readTVar read
    tryHead <- readTVar listHead
    case tryHead of
      TNil      -> return True
      TCons _ _ -> return False
```

Alternative Kanalimplementierung

- Statt verzeigerter Struktur: Verwende normale Liste
- Funktioniert, da mit `retry` auf beliebige Bedingungen gewartet werden kann

```
type TList a = TVar [a]
neueTList :: STM (TList a)
neueTList = newTVar []
schreibeTList :: TList a -> a -> STM ()
schreibeTList l val = do xs <- readTVar l
                          writeTVar l (xs ++ [val])
leseTList :: TList a -> STM a
leseTList l = do xs <- readTVar l
                 case xs of
                   [] -> retry
                   (val:xs) -> do writeTVar l xs
                                return val
```

Alternative Kanalimplementierung (2)

- Duplizieren wird durch die Implementierung mit Listen nicht unterstützt
- Ineffizienz beim Schreiben: Lineare Laufzeit wegen `++`
- Abhilfe: Statt einer Liste `xs`, merkt man sich zwei Listen `as` und `bs`, wobei stets gelten soll, dass die eigentliche Liste `xs` durch `as++(reversebs)` berechnet werden kann.

Alternative Kanalimplementierung (3)

```
type TListF a = (TVar [a],TVar [a])

neueTListF :: STM (TListF a)
neueTListF = do l <- newTVar []
              s <- newTVar []
              return (l,s)

schreibeTListF :: TListF a -> a -> STM ()
schreibeTListF (l,s) val = do bs <- readTVar s
                              writeTVar s (val:bs)
```

Alternative Kanalimplementierung (4)

```
leseTListF :: TListF a -> STM a
leseTListF (l,s) =
  do as <- readTVar l
     case as of
       (val:as') -> do writeTVar l as'
                      return val
       [] -> do bs <- readTVar s
                case bs of
                  [] -> retry
                  _ -> do let (val:as) = reverse bs
                           writeTVar l as
                           writeTVar s []
                           return val
```

Amortisierter Aufwand ist konstant.

Wiederholung: STM Haskell - API

Transaktionale Variablen

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

Transaktionen komponieren

```
return     :: a -> STM a
(>>=)     :: STM a -> (a -> STM b) -> STM b
orElse     :: STM a -> STM a -> STM a
retry      :: STM a
```

Transaktionen ausführen

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

Implementierung von STM-Haskell im GHC

- Wir beschreiben die interne Implementierung im Glasgow Haskell Compiler
- Insbesondere: Wie werden Konflikte erkannt?
- Wann darf eine Transaktion committen?

Transaktions-Log

- Pro Thread wird ein Transaktions-Log verwaltet
- Transaktions-Log: Tabelle, die enthält:
 - Welche TVars wurden geschrieben, gelesen, erzeugt?
 - Gelesener Wert der TVar
 - Neuer (zu schreibender) Wert der TVar
- Transaktions-Log ist **gültig**, wenn gelesene Werte mit den aktuellen Werten der TVars übereinstimmen.
- Gleichheit dabei: Pointer-Gleichheit.

Transaktion ausführen

- Leeres Transaktions-Log anlegen
- Lese-, Schreibe-, Erzeuge-Operationen im Log notieren
- Am Ende der Transaktion: Gültigkeit des Log prüfen
- Wenn Log gültig, dann schreibe neue Werte in die TVars
- Wenn Log ungültig, dann starte neu (verwerfe Log)

Beachte:

- Gültigkeit prüfen und schreiben passiert atomar bez. der anderen Transaktionen.
- `retry` und `orElse` noch nicht berücksichtigt

Beispiel zur Konflikterkennung

saldo :: TVar Int

785

Thread 1

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local + 1)  
  )
```

TVar	gelesen	geschrieben
saldo	7	8

Transaction log

Thread 2

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local - 3)  
  )
```

TVar	gelesen	geschrieben
saldo	78	45

Transaction log

Abort & Restart

Experimente zur Konflikterkennung (1)

```
demonstrateConflict mode n = do  
  mutex <- newMVar ()  
  tv <- newTVarIO 0  
  ids <- mapM async [transaction mutex mode i tv | i <- [1..n]]  
  sequence_ [wait i | i <- ids]
```

Experimente zur Konflikterkennung (2)

```
transaction mutex mode i tvar = do
count <- newIORef 0
atomically $ do
unsafeIOToSTM (do
  modifyIORef count (+1)
  c <- readIORef count
  printS mutex $ "transaction " ++ show i ++ " starts for the "
    ++ show c ++ ". time")
val <- readTVar tvar
let val' = val
let computeSomething k = if k > i*1000 then 0
  else computeSomething (k+1)
seq (computeSomething 0) -- waste some time
  (case mode of
    Different -> writeTVar tvar i
    OldVal    -> writeTVar tvar val
    PointerCopy -> writeTVar tvar val'
    SameVal   -> writeTVar tvar 0
    SameEx    -> writeTVar tvar (1-1))
printS mutex ("transaction " ++ show i ++ " finished")
```

Implementierung mit retry

- Jede TVar hat eine assoziierte Warteschlange von Threads
- Wird retry ausgeführt, so hängt sich der entsprechende Thread in die Wartelisten aller gelesenen TVars ein und blockiert.
- Wenn anderer Thread committed, so entblockiert er alle Threads in den Warteschlangen der TVars, die er beschreibt. Diese prüfen, ob sich die Werte gegenüber den gelesenen nun geändert haben und starten dann neu, oder warten erneut, wenn die Werte noch gleich sind.

Experimente zur Konflikterkennung (3)

Modus	Verhalten (ghci)	Verhalten (ghc -O0)	Verhalten (ghc -O1)	Verhalten (ghc -O1 -threaded und +RTS -N)
Different	Konflikte	Konflikte	Konflikte	Konflikte
OldVal	keine Konflikte (merkwürdiges Hängen)	keine Konflikte	keine Konflikte	Konflikte
SameVal	Konflikte	Konflikte	keine Konflikte	Konflikte
SameEx	Konflikte	Konflikte	keine Konflikte	Konflikte
PointerCopy	Konflikte	keine Konflikte	keine Konflikte	Konflikte

Implementierung mit orElse

- Transaktions-Log ist ein Stack der vorher beschriebenen Tabellen, für die neuen zu schreibenenden Werte, die gelesenen Werte werden alle (ohne Stack) aufgehoben
- orElse erzeugt neue Ebene im Stack
- Idee: Führt bei orElse $T_1 T_2$ die Transaktion T_1 zu retry, so wird die oberste Ebene des Stacks gelöscht, bevor T_2 gestartet wird.
- Für die Prüfung der Gültigkeit des Transaktions-Log werden alle gelesenen TVars geprüft.

Regelmäßiges Prüfen des Log

- GHC prüft das Transaktions-Log regelmäßig (nicht nur am Ende)
- Dadurch: Frühere Konflikterkennung
- Notwendig für semantisch korrekte Behandlung nichtterminierender Transaktionen!

Beispiel

```
i1 tv = atomically (  
  do  
    c <- readTVar tv  
    if c then  
      let loop i = do loop (i+1) in loop 0  
      else return ()  
  )  
  
i2 tv = atomically (writeTVar tv False)  
  
main = do tv <- atomically (newTVar True)  
         s <- newEmptyMVar  
         forkIO (i1 tv >> putMVar s ())  
         forkIO (i2 tv >> putMVar s ())  
         takeMVar s >> takeMVar s
```

Eigenschaften von Haskell's STM

- Isolation:** Strong (kein Zugriff auf Transaktionsvariablen außerhalb von Transaktionen)
- Granularität:** Wortbasiert
- Update:** Verzögert (Neue Werte werden ins Log geschrieben, erst beim commit in den Speicher)
- Concurrency Control:** optimistisch (meint keine Locks auf die Speicherplätze)
- Synchronisation:** Blockierend
- Conflict Detection:** Late (erst beim commit)
- Nested Transaction:** Nein (durch Typsystem verhindert)