Functional Programming

Software Transactional Memory

Jaak Randmets

Department of Computer Science
University of Tartu

Introduction

Concurrent programming

- Free lunch is over.
- Concurrent programming is difficult.
- Locks.
  - Too many, too few.
  - Wrong order.
  - Error recovery.

Introduction

Concurrent programming

- Free lunch is over.
- Concurrent programming is difficult.
- Locks.
  - Too many, too few.
  - Wrong order.
  - Error recovery.

Introduction

Concurrent programming

- Free lunch is over.
- Concurrent programming is difficult.
- Locks.
  - Too many, too few.
  - Wrong order.
  - Error recovery.

Introduction

Concurrent programming

- Free lunch is over.
- Concurrent programming is difficult.
- Locks.
  - Too many, too few.
  - Wrong order.
  - Error recovery.

Introduction

Concurrent programming

- Free lunch is over.
- Concurrent programming is difficult.
- Locks.
  - Too many, too few.
  - Wrong order.
  - Error recovery.

Introduction

Concurrent programming

- Free lunch is over.
- Concurrent programming is difficult.
- Locks.
  - Too many, too few.
  - Wrong order.
  - Error recovery.

Demotivating example

Bad

$$
\begin{array}{l}
\mathit{relocate}\ \mathit{t1}\ \mathit{t2}\ \mathit{k} = \mathbf{do} \\
\quad v \leftarrow \mathit{remove}\ \mathit{t1}\ \mathit{k} \\
\quad \mathit{insert}\ \mathit{t2}\ \mathit{k}\ \mathit{v}
\end{array}
$$

Better?

$$
\begin{array}{l}
\mathit{relocate}\ \mathit{t1}\ \mathit{t2}\ \mathit{k} = \mathbf{do} \\
\quad \mathit{lock}\ \mathit{t1}; \mathit{lock}\ \mathit{t2} \\
\quad v \leftarrow \mathit{remove}'\ \mathit{t1}\ \mathit{k} \\
\quad \mathit{insert}'\ \mathit{t2}\ \mathit{k}\ \mathit{v} \\
\quad \mathit{unlock}\ \mathit{t1}; \mathit{unlock}\ \mathit{t2}
\end{array}
$$

Software Transactional Memory

Definition

- Transaction is block of code that reads and writes memory.
- Execution of transaction is atomic and isolated
  - Atomicity: effects of executed block become visible to other threads all at once.
  - Isolation: action is completely unaffected by other threads.

Old idea

- Optimistic.
- Record every read and write to a local log.
- Log is validated.
  - If log is valid then transaction is committed.
  - otherwise the log is discard and transaction re-executed.

Software Transactional Memory

Definition

- Transaction is block of code that reads and writes memory.
- Execution of transaction is atomic and isolated
  - Atomicity: effects of executed block become visible to other threads all at once.
  - Isolation: action is completely unaffected by other threads.

Old idea

- Optimistic.
- Record every read and write to a local log.
- Log is validated.
  - If log is valid then transaction is committed.
  - otherwise the log is discard and transaction re-executed.

Software Transactional Memory

Definition

- Transaction is block of code that reads and writes memory.
- Execution of transaction is atomic and isolated
  - Atomicity: effects of executed block become visible to other threads all at once.
  - Isolation: action is completely unaffected by other threads.

Old idea

- Optimistic.
- Record every read and write to a local log.
- Log is validated.
  - If log is valid then transaction is committed.
  - otherwise the log is discard and transaction re-executed.

Software Transactional Memory

Definition

- Transaction is block of code that reads and writes memory.
- Execution of transaction is atomic and isolated
  - Atomicity: effects of executed block become visible to other threads all at once.
  - Isolation: action is completely unaffected by other threads.

Old idea

- Optimistic.
- Record every read and write to a local log.
- Log is validated.
  - If log is valid then transaction is committed.
  - otherwise the log is discard and transaction re-executed.

Software Transactional Memory

Definition

- Transaction is block of code that reads and writes memory.
- Execution of transaction is atomic and isolated
  - Atomicity: effects of executed block become visible to other threads all at once.
  - Isolation: action is completely unaffected by other threads.

Old idea

- Optimistic.
- Record every read and write to a local log.
- Log is validated.
  - If log is valid then transaction is committed.
  - otherwise the log is discard and transaction re-executed.

STM in Haskell

STM is a monad

data *STM a*
instance *Monad STM*

- Sequential composition.
- Do notation.
- No IO inside transaction.
- No STM action can be performed outside transaction.

STM in Haskell

Atomically and *TVars*

Executing transaction:

$$atomically :: STM\ a \to IO\ a$$

TVars:

$$\begin{aligned}
&\textbf{data}\ TVar\ a \\
&newTVar\ \ :: a \to STM\ (TVar\ a) \\
&readTVar\ \ :: TVar\ a \to STM\ a \\
&writeTVar :: TVar\ a \to a \to STM\ ()
\end{aligned}$$

NB!

$$(atomically\ M1) \gg (atomically\ M2) \not\equiv atomically\ (M1 \gg M2)$$

Example

Simple example

```
type Resource = TVar Int
putR :: Resource → Int → STM ()
putR r i = do
  v ← readTVar r
  writeTVar r (v + i)
main = do
  r ← atomically $ newTVar 0
  sequence_ ∘ replicate 10 ∘ forkIO ∘ atomically $ putR r 3
  threadDelay 1000
  n ← atomically $ readTVar r
  print n
```

STM in Haskell

Blocking and composing alternatives

$$retry :: STM\ a$$
$$orElse :: STM\ a \to STM\ a \to STM\ a$$

STM is instance of MonadPlus:

$$M1\ `orElse`\ (M2\ `orElse`\ M3) = (M1\ `orElse`\ M2)\ `orElse`\ M3$$
$$retry\ `orElse`\ M = M$$
$$M\ `orElse`\ retry = M$$

Example

Blocking transaction

```
getR :: Resource → Int → STM ()
getR r i = do
  v ← readTVar r
  if (v < i)
    then retry
    else writeTVar r (v − i)
```

check

```
check :: Bool → STM ()
check True = return ()
check Flase = retry
```

Exceptions

throw and *catch*

$$
\begin{aligned}
&\mathit{throw} && :: \mathit{Exception} \to a \\
&\mathit{catch} && :: \mathit{IO}\ a \to (\mathit{Exception} \to \mathit{IO}\ a) \to \mathit{IO}\ a \\
&\mathit{catchSTM} && :: \mathit{STM}\ a \to (\mathit{Exception} \to \mathit{STM}\ a) \to \mathit{STM}\ a
\end{aligned}
$$

Exceptions allow values to "leak" out of STM.

A leak

```
tv ← atomically $ newTVar "hello"
Control.Exception.catch (atomically $ do
  updateTVar tv (++", world!")
  s ← readTVar tv
  throw (AssertionFailed s)) print
atomically (readTVar tv) >>= print
```

Channels and MVars

TChan

```
data TChan a
newTChan      :: STM (TChan a)
readTChan     :: TChan a → STM a
writeTChan    :: TChan a → a → STM ()
isEmptyTChan  :: TChan a → STM Bool
```

MVar

```
newEmptyMVar :: STM (MVar a)
takeMVar     :: MVar a → STM a
putMVar      :: MVar a → a → STM ()
```

Examples

MVar implementation

```
type MVar a = TVar (Maybe a)
newEmptyMVar = newTVar Nothing
takeMVar mv = do
  v ← readTVar mv
  case v of
    Nothing → retry
    Just val → writeTVar mv Nothing ≫ return val
putMVar mv val = do
  v ← readTVar mv
  case v of
    Nothing → writeTVar mv (Just val)
    Just _  → retry
```

Examples

Producers/consumer

```
main = do
  tc ← atomically $ newTChan
    -- producers
  sequence_ ∘ replicate 10 ∘ forkIO $ do
    forM_ [1..100] $ λi → do
      threadDelay 2
      atomically $ writeTChan tc i
    putStrLn "work, work!"
    -- consumer
  forkIO ∘ forever $ do
    threadDelay $ 1000 * 25
    x ← atomically $ readTChan tc
    putStr $ (show x) ++ "  "
```

Examples

Useful functions

Applying function to TVar:

$$updateTVar :: TVar\ a \to (a \to a) \to STM\ ()$$
$$updateTVar\ tv\ f = readTVar\ tv \gg\!= writeTVar\ tv \circ f$$

Failing transaction:

$$orZero :: (MonadPlus\ m) \Rightarrow STM\ a \to STM\ (m\ a)$$
$$orZero\ st = (st \gg\!= return \circ return)\ `orElse`\ return\ mzero$$

Specialising *orZero*:

$$orNothing :: STM\ a \to STM\ (Maybe\ a)$$
$$orNothing = orZero$$

Examples

Useful functions

Merging transactions:

```
merge :: [STM a] → STM a
merge = foldr1 orElse
```

Choosing transaction and *IO* action pair:

```
choose :: [(STM a, a → IO ())] → IO ()
choose choices = (atomically $ merge actions) >>= id
  where
    actions :: [STM (IO ())]
    actions = [guard >>= return ∘ rhs | (guard, rhs) ← choices]
```