Software Transactinal Memory

Henrik Nilsson

University of Nottingham, UK

This Lecture

- Some problems with standard approaches to synchronisation
- Software Transactional Memory (STM)
- Haskell used for illustration throughout
- We will also see that STM and pure functional programming is a particularly good match
- We will start with a quick overview of concurrent programming in Haskell.

Concurrent Programming in Haskell

Primitives for concurrent programming provided as operations of the ***IO monad***. Excerpts:

```
forkIO       :: IO () -> IO ThreadId
killThread   :: ThreadId -> IO ()
threadDelay  :: Int -> IO ()
newMVar      :: a -> IO (MVar a)
newEmptyMVar :: IO (MVar a)
putMVar      :: MVar a -> a -> IO ()
takeMVar     :: MVar a -> IO a
```

The IO Monad??? (1)

The IO Monad??? (1)

- Haskell uses ***monads*** as a “bridge” between the pure functional world and the world of input/output, state, and other ***effects***.

The IO Monad??? (1)

- Haskell uses ***monads*** as a “bridge” between the pure functional world and the world of input/output, state, and other ***effects***.
- For the purpose of this talk, think about a monadic value of type `m a` as a ***computation*** in the monad `m` returning a value of type `a` described by a sequence of ***monadic actions*** or “commands”.

The IO Monad??? (2)

- Each monad embodies a particular set of effects.

The IO Monad??? (2)

- Each monad embodies a particular set of effects.
- Computations may be ***composed*** into larger computations, but . . .

The IO Monad??? (2)

- Each monad embodies a particular set of effects.
- Computations may be ***composed*** into larger computations, but . . .
- . . . only when a computation is ***"run"*** are the actions and their side effects actually carried out.

The IO Monad??? (2)

- Each monad embodies a particular set of effects.
- Computations may be ***composed*** into larger computations, but . . .
- . . . only when a computation is ***"run"*** are the actions and their side effects actually carried out.

Key point: ***disciplined use of effects***: types account for precisely which effects can occur where.

Concurrency primitives again

Let us revisit the IO concurrency primitives again in the light of what we now know about monads:

```
forkIO       :: IO () -> IO ThreadId
killThread   :: ThreadId -> IO ()
threadDelay  :: Int -> IO ()
newMVar      :: a -> IO (MVar a)
newEmptyMVar :: IO (MVar a)
putMVar      :: MVar a -> a -> IO ()
takeMVar     :: MVar a -> IO a
```

MVars

`MVars`

- The fundamental synchronisation mechanism is the ***`MVar`*** (“em-var”).

`MVars`

- The fundamental synchronisation mechanism is the ***`MVar`*** (“em-var”).
- An `MVar` is a “one-item box” that may be ***empty*** or ***full***.

`MVars`

- The fundamental synchronisation mechanism is the ***`MVar`*** (“em-var”).
- An `MVar` is a “one-item box” that may be ***empty*** or ***full***.
- Reading (`takeMVar`) and writing (`putMVar`) are ***atomic*** operations:

MVars

- The fundamental synchronisation mechanism is the ***`MVar`*** (“em-var”).
- An `MVar` is a “one-item box” that may be ***empty*** or ***full***.
- Reading (`takeMVar`) and writing (`putMVar`) are ***atomic*** operations:
  - Writing to an empty `MVar` makes it full.

`MVars`

- The fundamental synchronisation mechanism is the ***`MVar`*** (“em-var”).
- An `MVar` is a “one-item box” that may be ***empty*** or ***full***.
- Reading (`takeMVar`) and writing (`putMVar`) are ***atomic*** operations:
  - Writing to an empty `MVar` makes it full.
  - Writing to a full `MVar` blocks.

MVars

- The fundamental synchronisation mechanism is the ***MVar*** (“em-var”).
- An `MVar` is a “one-item box” that may be ***empty*** or ***full***.
- Reading (`takeMVar`) and writing (`putMVar`) are ***atomic*** operations:
  - Writing to an empty `MVar` makes it full.
  - Writing to a full `MVar` blocks.
  - Reading from an empty `MVar` blocks.

MVars

- The fundamental synchronisation mechanism is the ***MVar*** ("em-var").
- An `MVar` is a "one-item box" that may be ***empty*** or ***full***.
- Reading (`takeMVar`) and writing (`putMVar`) are ***atomic*** operations:
  - Writing to an empty `MVar` makes it full.
  - Writing to a full `MVar` blocks.
  - Reading from an empty `MVar` blocks.
  - Reading from a full `MVar` makes it empty.

Example: Basic Synchronization (1)

```
module Main where

import Control.Concurrent

countFromTo :: Int -> Int -> IO ()
countFromTo m n
    | m > n     = return ()
    | otherwise = do
        putStrLn (show m)
        countFromTo (m+1) n
```

Example: Basic Synchronization (2)

```
main = do
    start <- newEmptyMVar
    done  <- newEmptyMVar
    forkIO $ do
        takeMVar start
        countFromTo 1 10
        putMVar done ()
    putStrLn "Go!"
    putMVar start ()
    takeMVar done
    (countFromTo 11 20)
    putStrLn "Done!"
```

Example: Unbounded Buffer (1)

```
module Main where

import Control.Monad (when)
import Control.Concurrent

newtype Buffer a =
    Buffer (MVar (Either [a] (Int, MVar a)))

newBuffer :: IO (Buffer a)
newBuffer = do
    b <- newMVar (Left [])
    return (Buffer b)
```

Example: Unbounded Buffer (2)

```
readBuffer :: Buffer a -> IO a
readBuffer (Buffer b) = do
    bc <- takeMVar b
    case bc of
        Left (x : xs) -> do
            putMVar b (Left xs)
            return x
        Left []       -> do
            w <- newEmptyMVar
            putMVar b (Right (1,w))
            takeMVar w
        Right (n,w)   -> do
            putMVar b (Right (n + 1, w))
            takeMVar w
```

Example: Unbounded Buffer (3)

Why isn't `Buffer` simply defined as

```
newtype Buffer a = Buffer [a]
```

?

Example: Unbounded Buffer (3)

Why isn't `Buffer` simply defined as

```
newtype Buffer a = Buffer [a]
```

?

Hint: What would happen if e.g. an attempt is made to read from an empty buffer?

Example: Unbounded Buffer (4)

```
writeBuffer :: Buffer a -> a -> IO ()
writeBuffer (Buffer b) x = do
    bc <- takeMVar b
    case bc of
        Left xs ->
            putMVar b (Left (xs ++ [x]))
        Right (n,w) -> do
            putMVar w x
            if n > 1 then
                putMVar b (Right (n - 1, w))
             else
                putMVar b (Left [])
```

Example: Unbounded Buffer (5)

The buffer can now be used as a channel of communication between a set of “writers” and a set of “readers”. E.g.

```
main = do
    b <- newBuffer
    forkIO (writer b)
    forkIO (writer b)
    forkIO (reader b)
    forkIO (reader b)
    ...
```

Example: Unbounded Buffer (6)

```
reader :: Buffer Int -> IO ()
reader n b = rLoop
    where
        rLoop = do
            x <- readBuffer b
            when (x > 0) $ do
                putStrLn (n ++ ": " ++ show x)
                rLoop
```

Compositionality? (1)

Suppose we would like to read two ***consecutive*** elements from a buffer b?

That is, ***sequential composition***.

Would the following work?

```
x1 <- readBuffer b
x2 <- readBuffer b
```

Compositionality? (2)

What about this?

```
mutex <- newMVar ()
...
takeMVar mutex
x1 <- readBuffer b
x2 <- readBuffer b
putMVar mutex ()
```

Compositionality? (3)

Suppose we would like to read from ***one of two*** buffers.

That is, ***composing alternatives***.

Compositionality? (3)

Suppose we would like to read from ***one of two*** buffers.

That is, ***composing alternatives***.

Hmmm. How do we even begin?

Compositionality? (3)

Suppose we would like to read from ***one of two*** buffers.

That is, ***composing alternatives***.

Hmmm. How do we even begin?

- No way to attempt reading a buffer without risking blocking.

Compositionality? (3)

Suppose we would like to read from ***one of two*** buffers.

That is, ***composing alternatives***.

Hmmm. How do we even begin?

- No way to attempt reading a buffer without risking blocking.
- We have to change or enrich the buffer implementation. E.g. add a `tryReadBuffer` operation, and then repeatedly poll the two buffers in a tight loop. Not so good!

Software Transactional Memory (1)

- Operations on shared mutable variables grouped into ***transactions***.
- A transaction either succeeds or fails in its ***entirety***. I.e., ***atomic*** w.r.t. other transactions.
- Failed transactions are automatically ***retried*** until they succeed.
- ***Transaction logs***, which records reading and writing of shared variables, maintained to enable transactions to be validated, partial transactions to be rolled back, and to determine when worth trying a transaction again.

Software Transactional Memory (2)

- ***No locks!*** (At the application level.)

STM and Pure Declarative Languages

- STM perfect match for ***purely declarative languages***:
  - reading and writing of shared mutable variables explicit and relatively rare;
  - most computations are pure and need not be logged.
- Disciplined use of effects through monads a ***huge*** payoff: easy to ensure that ***only*** effects that can be undone can go inside a transaction.

  (Imagine the havoc arbitrary I/O actions could cause if part of transaction: How to undo? What if retried?)

The STM monad

The software transactional memory abstraction provided by a monad `STM`. ***Distinct from IO!*** Defined in `Control.Concurrent.STM`.

Excerpts:

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

Example: Buffer Revisited (1)

Let us rewrite the unbounded buffer using the STM monad:

```
module Main where

import Control.Monad (when)
import Control.Concurrent
import Control.Concurrent.STM

newtype Buffer a = Buffer (TVar [a])

newBuffer :: STM (Buffer a)
newBuffer = do
    b <- newTVar []
    return (Buffer b)
```

Example: Buffer Revisited (2)

```
readBuffer :: Buffer a -> STM a
readBuffer (Buffer b) = do
    xs <- readTVar b
    case xs of
        []         -> retry
        (x : xs’) -> do
            writeTVar b xs’
            return x

writeBuffer :: Buffer a -> a -> STM ()
writeBuffer (Buffer b) x = do
    xs <- readTVar b
    writeTVar b (xs ++ [x])
```

Example: Buffer Revisited (3)

The main program and code for readers and writers can remain unchanged, except that STM operations must be carried out ***atomically***:

```
main = do
    b <- atomically newBuffer
    forkIO (writer b)
    forkIO (writer b)
    forkIO (reader b)
    forkIO (reader b)
    ...
```

Example: Buffer Revisited (4)

```
reader :: Buffer Int -> IO ()
reader n b = rLoop
    where
        rLoop = do
            x <- atomically (readBuffer b)
            when (x > 0) $ do
                putStrLn (n ++ ": " ++ show x)
                rLoop
```

Why shouldn't `atomically` be part of the definition of `readBuffer`?

Composition (1)

STM operations can be ***robustly composed***. That's the reason for making `readBuffer` and `writeBuffer` STM operations, and leaving it to client code to decide the scope of atomic blocks.

Example, sequential composition: reading two consecutive elements from a buffer b:

```
atomically $ do
    x1 <- readBuffer b
    x2 <- readBuffer b
    ...
```

Composition (2)

Example, composing alternatives: reading from one of two buffers `b1` and `b2`:

```
x <- atomically $
         readBuffer b1
         ‘orElse‘ readBuffer b2
```

The buffer operations thus composes nicely. No need to change the implementation of any of the operations!

Reading

- Koen Claessen. A Poor Man's Concurrency Monad. *Journal of Functional Programming*, 9(3), 1999.
- Wouter Swierstra and Thorsten Altenkirch. Beauty in the Beast: A Functional Semantics for the Awkward Squad. In *Proceedings of Haskell'07*, 2007.
- Tim Harris, Simon Marlow, Simon Peyton Jones, Maurice Herlihy. Composable Memory Transactions. In *Proceedings of PPoPP'05*, 2005
- Simon Peyton Jones. Beautiful Concurrency. Chapter from *Beautiful Code*, ed. Greg Wilson, O'Reilly 2007.