Utrecht University

B3CC: Concurrency

08: Software Transactional Memory (2)

Tom Smeding

Recap

- An approach to implementing *atomic blocks*
  - Effects become visible to other threads all at once
  - Actions within an `atomically :: STM a -> IO a` block are executed isolated from all other threads
  - Execute optimistically: roll-back changes and retry when a conflict is detected
  - Offers *composable* blocking and atomicity

Recap

```
import Control.Concurrent.STM

data STM a
instance Monad STM

atomically :: STM a -> IO a
retry      :: STM a
orElse     :: STM a -> STM a -> STM a

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

(and a few more, but we won't discuss those here)

Correction

- About the progress property of STM

STM as a building block (II)

Concurrent Map

Key-value map

- The goal:
  - A key-value map that can be accessed concurrently by multiple threads
  - Basic interface:

```
data CMap k v

insert :: Ord k => k -> v -> CMap k v -> CMap k v
lookup :: Ord k => k -> CMap k v -> Maybe v
```

Option #1

- A regular (pure) key-value map in a mutable box
  - Simple, safe
  - No concurrency!

```
import Control.Concurrent.MVar
import qualified Data.Map as M

data CMap k v = CMap (MVar (M.Map k v))

insert :: Ord k => k -> v -> CMap k v -> IO ()
lookup :: Ord k => k -> CMap k v -> IO (Maybe v)
```

Option #2

- A pure map in a box, but this time using STM
  - Safe concurrent lookup
  - Insertion updates the entire tree (*all* other threads must retry)

```
import Control.Concurrent.STM
import qualified Data.Map as M

data CMap k v = CMap (TVar (M.Map k v))

insert :: Ord k => k -> v -> CMap k v -> STM ()
lookup :: Ord k => k -> CMap k v -> STM (Maybe v)
```

Option #3

- A pure map with mutable values
  - Allows values to be read and adjusted (mutated) concurrently
  - Fixed key set

```
import Control.Concurrent.STM
import qualified Data.Map as M

data CMap k v = CMap (M.Map k (TVar v))

adjust :: Ord k => (v -> v) -> k -> CMap k v -> STM ()
lookup :: Ord k => k -> CMap k v -> STM (Maybe v)
```

Option #4

- Implement the data structure ourselves
  - Goal: Fully concurrent insertion and lookup
  - Updates to disjoint parts of the tree do not conflict with each other

```
data CMap k v = CMap (TVar (Node k v))
data Node k v
  = Bin k (TVar v) (CMap k v) (CMap k v)
  | Tip

insert :: Ord k => k -> v -> CMap k v -> STM ()
lookup :: Ord k => k -> CMap k v -> STM (Maybe v)
adjust :: Ord k => (v -> v) -> k -> CMap k v -> STM ()
```

Option #4

- Lookup a value in the map
  - Standard recursive traversal
  - Try to implement `insert`!
    - Minimise the number of `writeTVar`!

```haskell
data CMap k v = CMap (TVar (Node k v))
data Node k v
  = Bin k (TVar v) (CMap k v) (CMap k v)
  | Tip
```

```haskell
lookup :: Ord k => k -> CMap k v -> STM (Maybe v)
lookup key (CMap ref) = readTVar ref >>= go
  where
    go Tip              = return Nothing
    go (Bin k v l r) =
      case compare key k of
        LT -> lookup key l
        GT -> lookup key r
        EQ -> Just <$> readTVar v
```

Summary

What can we not do with STM?

- STM offers *composable* blocking and atomicity
  - Concurrent programming without locks!
- But, there are also things that it can *not* do compared to using locks
  - Fairness: all blocked threads are woken up when a `TVar` changes
  - No progress guarantee
  - Threads can not communicate *that* they are blocking

Performance considerations

- `atomically` works by accumulating a log of `writeTVar` and `readTVar` operations; this has consequences:
  - Discarding the effects of the transaction is easy: delete the log
  - Each `readTVar` must traverse the log to see if it was written by an earlier `writeTVar`: $O(n)$
  - A transaction that called `retry` is woken up whenever one of the `TVars` in its read set changes: $O(n)$
  - A long running transaction can re-execute indefinitely because it is repeatedly aborted by shorter transactions: starvation
- Most abstractions have a runtime cost…

IORefs as a building block (II)

Lockfree concurrent queue

Unbounded queue

- The goal:
  - As before, but implement a lock-free queue
  - Creating a new empty queue is similar to before, but both ends point to `Cons` cell as a sentinel value

```
data Queue a =
  Queue (IORef (List a))
        (IORef (List a))

data List a = Null
            | Cons a (IORef (List a))
```

This is the Michael–Scott queue: https://www.cs.rochester.edu/research/synchronization/pseudocode/queues.html

enqueue

```
data Queue a =
  Queue (IORef (List a))
        (IORef (List a))

data List a = Null
            | Cons a (IORef (List a))
```

- To add an element to the queue
  1. Create the new cell holding the value, with next pointing to `Null`
  2. Keep trying until done:
     1. Read the tail from the queue
     2. Read the next node from the tail
     3. Does the next node point to Null?
        - The tail pointed to the last node: try to link our node at the end of the queue (CAS); otherwise
        - Somebody else beat us extending the tail; help out by trying to swing the tail to the next node (CAS)
  3. Try to swing the tail to the inserted node; this might fail but that is okay (CAS)

dequeue

- To remove an element from the queue

```
data Queue a =
  Queue (IORef (List a))
        (IORef (List a))

data List a = Null
            | Cons a (IORef (List a))
```

1. Read the head and tail pointers
2. Are head and tail equal?
   - Empty, or outdated tail?
     Read head.next; if `Cons`, advance tail (CAS) and try again. If `Null`, queue is empty.
3. Read head.next.value
4. Advance head (CAS); if this fails, another thread has already claimed this node, so try again

tot ziens

Photo by Simon Oliehoek

Extra slides

- The Next Mainstream Programming Language: A Game Developer's Perspective (2005)
  - https://groups.csail.mit.edu/cag/crg/papers/sweeney06games.pdf
- Beyond Functional Programming: The Verse Programming Language (2022)
  - https://simon.peytonjones.org/assets/pdfs/haskell-exchange-22.pdf