Composable Shared Memory Transactions
Lecture 20-2

April 3, 2008

This was actually the 21st lecture of the class, but I messed up the naming of subsequent notes files so I'll just call this one 20-2.

This lecture is based on *Composable Memory Transactions,* by Harris, Marlow, Peyton Jones, and Herlihy, ACM PPoPP, June 15-17, 2005.

There are a number of issues related to concurrency and transactions that were not solved in the STM paper that we discussed last time.

1. As we noted last time no mechanism is provided to wait for some condition to become true. Thus data structures like bounded buffers, resource managers, etc. can only be implemented by busy waiting.
2. The "usual" ways of concurrent programming using locks and conditions (e.g. the Java model) do not support *composition* of actions. A client cannot use a "get the first element" method implemented by a queue to get the first two elements by calling the method twice in the presence of concurrency – another thread make sneak in between the two calls.
3. How should external actions be handled? If a "withdraw $100" transaction aborts and restarts you don't want the ATM to spit out $100 on each try. Existing DB systems have ad hoc mechanisms to help solve this problem.
4. Finally, what if we want to do either of two actions (whichever one becomes enabled first), a form of non-deterministic choice. This is what the `select` operation does in your programming assignment.

This paper presents elegant solutions to all these issues in a language called Haskell. It is strongly, statically typed, purely functional, and lazy. Being purely functional and also doing IO is a challenge – IO does not satisfy our usual understanding of what it means to be functional. In Haskell this problem is solved by so-called monadic IO, and this solution will also be what we need to elegantly solve the isolation of external actions from transactions (issue three above).

Monadic IO

Haskell clearly separates IO *evaluation of functions* from *performing IO actions.* The separation is enforced by the type system. A Haskell program contains a unique `main` function that must have type `IO a` for some type `a`. A *value* of type `IO a` is called an *IO action.* Until it is *performed* an IO action is just a value. Actions can be composed from other actions using `do` notation that allows results of one IO action (input) to be made available to other IO actions, and to use inputs as function arguments.

```
main :: IO ()      // main is a value with type unit IO action (i.e it pro
                   // no input to other IO actions; this is just a type de
main = putChar 'x' // putChar is a function that takes a character and re
                    // an IO action
```

When a program is run the IO action of its `main` is performed. (Note that the IO action can be compound, composed of many other IO actions.)

```
forkIO :: IO a -> IO ThreadId // ForkIO creates a concurrent thread
main = do { forkIO (print 'x'); print 'y' }
```

`do` composes IO actions sequentially and in order. `do { print 'x'; print 'y' }` is an IO action that first prints x then y. Using `forkIO` allows the IO action of its argument to run concurrently with performance of the forking action. Again, notice that the type system enforces that IO actions are only performed as a result of performing the main action. *Evaluation* of functions does not result in *performing* IO actions so there is strong containment of IO side effects of a program to only the IO actions.

STM in Haskell

The same linguistic machinery used for IO can also be applied to transactions: we end up with now 3 boxes: transactional actions, IO actions, and function evaluation, and again the type system allows only carefully designed interactions between them.

The idea is best illustrated with an example. The paper calls it a resource manager, but it has operations that are familiar to us as semaphores.

```
type Resource = TVar Int // TVar is a transactional variable; TVar Int is
                         // transactional integer
putR :: Resource -> Int -> STM () // put a number of resources into a Res
                                  // variable; result is an STM action
putR r i = do { v <- readTVar r; writeTVar r (v+i) }
```

An STM action (such as the result of putR) is, like an IO action, merely a value. It doesn't do anything until it is performed. Unlike an IO action, an STM action is performed using the function `atomic` which converts an STM action to an IO action that when performed performs the STM action.

```
atomic :: STM a -> IO a
main = do { ...; atomic (putR r 3) ; ... }
```

`atomic` is a primitive of Haskell's STM system. When the action is a performed it

- checks that no conflicting updates have been committed to TVars read by the transaction; if a conflict is detected it repeats the transaction from the beginning
- atomically commits the transaction's changes to TVars written by the transaction

The type system guarantees that IO actions cannot be performed inside a transaction so no visible side effects can occur while the transaction is still abortable solving issue 3 above. Also STM actions cannot be performed outside of an IO action, (so STM actions occur in a definite order wrt to IO actions – something that is not true of the "outside" purely functional world.)

Waiting (issue 1 above) and Composability (issue 2)

Waiting for another thread to make some (abstract) condition true is not supported in our previous discussion of STM. Composable memory transactions address waiting using `retry`, an STM action that when performed aborts the transaction, waits for at least one TVar read by the transaction to be updated, then starts the transaction from the beginning. Example:

```
getR :: Resource -> Int -> STM ()
getR r i = do { v <- readTVar r;
                if (v<i) then retry
                else writeTVar r (v-i) }
```

Note that the transaction aborted by the `retry` is not merely the enclosing `do {...}`, but everything in the `atomic` that uses `getR`, so in

```
atomic (do {getR r1 3; getR r2 7})
```

failure to get 7 units from r2 gives up the 3 units obtained from r1 and then we try again. The claim is that STM transactions compose serially in a way that monitor-based concurrent actions do no.

Finally, note that this approach raises questions of efficiency. `retry` tries again when *any* referenced TVar changes in any way. So executions may be tried that are doomed to fail – for example if more resources are taken from r2 in the above example, or if changes are made to r1. The authors note that as compensation the mechanisms are easy to use correctly in comparison to conventional synchronization mechanisms.

Choice (issue 4)

Finally, many conventional concurrency systems (but not all) and the STM mechanisms do not provide good support for composable choice. This paper introduces `orElse` as a primitive for composable choice.

```
atomic (getR r1 3 `orElse` getR r2 7)
```

means try to perform (getR r1 3). If it succeeds we're done; otherwise (that is, it retries) try to perform (getR r2 7). If it succeeds we're done; otherwise (it retries) retry the whole transaction waiting on the union of the referenced variables from both operands of `orElse`. (The backquotes are just a Haskell-ism that allows use of an arbitrary function as an infix operator.)

With `orElse` it is possible to implement non-blocking operations in terms of blocking ones and vice-versa.

```
nonBlockGetR :: Resource -> Int -> STM Bool
nonBlockGetR r i = do {getR r i; return True} `orElse` {return False}
```

The idea of `nonBlockGetR` is that if getR would block it returns `False` indicating failure to get the resource, but it doesn't block. Similarly, given `nonBlockGetR` you can implement `blockGetR` which has the same behavior as the original `getR` using:

```
blockGetR r i = do { s <- nonBlockGetR r i; if s then return () else retr
```

Note that:

```
retry `orElse` M === M
M `orElse` retry === M
```

for any `M`.

Section 4 of the paper contains additional examples. Finally a few notes on implementation.

`atomic` when performed, allocates a stack from containing a new log, then performs its contained actions. When the actions return to the frame containing the log, the log is validated by checking for conflicting changes to referenced TVars and committed or aborted accordingly. References to TVars always check the log first so a transaction sees its own changes while it is executing.

`retry` when performed unwinds back to the atomic frame, then waits for a change to one of the variables that the transaction has referenced. Note that it knows this because all references have to be recorded in the log associated with the atomic frame.

`orElse` when performed starts a nested transaction – a separate log. Note that each of the subtransactions of an `orElse` may itself involve further composition. When both branches retry `orElse` handles promote the logged variables to the containing transaction.