Open memory transactions in Haskell

Marino Miculan, Marco Peressotti and Andrea Toneguzzo

MADS, Department of Mathematics and Computer Science, University of Udine, Italy
`marino.miculan@uniud.it, marco.peressotti@uniud.it, andrea.toneguzzo@spes.uniud.it`

Transactional memory (TM) has emerged as a promising mechanism to replace locks [1,3]. The basic idea is to mark blocks of code as *atomic*; then, execution of each block will appear either if it was executed instantaneously, at some unique point in time, or, if aborted, as if it did not execute at all. This is obtained by means of *optimistic* executions: the blocks are allowed to run concurrently, and eventually if an interference is detected a transaction is restarted and its effects are rolled back. Differently from lock-based concurrency control mechanisms, transactions are composable and ensure absence of deadlocks and priority inversions, automatic roll-back on exceptions, and increased concurrency. Moreover, each transaction can be viewed in isolation as a *single-threaded* computation, significantly reducing programmer's burden.

However, in multi-threaded programming different transactions may need to interact and exchange data *before* reaching the commit phase. In this situation, transaction isolation is a severe shortcoming. A simple example is a synchronization (rendezvous) between threads belonging to different transactions. A naive attempt would be to use two semaphores `c1`, `c2` as shown aside. Unfortunately, this solution does not work: any admissible execution requires an interleaved scheduling between the two transactions, thus violating isolation; hence, the transactions deadlock as none of them can progress. It is important to notice that this deadlock arises because synchronization occurs between threads in *different* transactions; in fact, the solution above works for threads *outside* transactions, or within the *same* transaction.

```
// Party1            // Party2
atomically {         atomically {
  // code before       // code before
  up(c1);              down(c1);
  down(c2);            up(c2);
  // code after        // code after
}                    }
```

In order to overcome this limitation, we propose a programming model for *safe, data-driven* interactions between memory transactions. The key observation is that *atomicity* and *isolation* should be seen as two disjoint computational aspects:

- an atomic *non-isolated* block of code is executed "all-or-nothing", but its execution can overlap that of others and *uncontrolled* access to shared data is allowed;
- an *isolated* block of code is intended to be executed "as it were the only one" (i.e., in mutual exclusion with other threads), but no rollback on errors/exceptions is provided.

Thus, a "normal" block of code is neither atomic nor isolated; a mutex block (like Java *synchronized* methods) is isolated but not atomic; and a usual transaction is a block which is both atomic and isolated. Our claim is that atomic non-isolated blocks can be fruitfully used for implementing safe composable interacting transactions, henceforth called *open transactions.*

In this model, a transaction is composed by several threads, called *participants*, which can cooperate on shared data. A transaction commits when all its participants commit, and aborts if any thread aborts. Threads participating to different transactions *can* access to shared data, but when this happens the transactions are transparently *merged* into a single one. For instance, the two transactions of the rendezvous example above would automatically merge becoming the same transaction, hence the two threads can synchronize and proceed. Thus, this model relaxes the isolation requirement still guaranteeing atomicity and consistency; moreover, it allows for *loosely-coupled* interactions since transaction merging is driven only by run-time accesses to shared data, without any explicit coordination among the participants beforehand.

```
type ITM a
type OTM a
-- t is a placeholder for ITM or OTM --

-- Sequencing, do notation ------------
(>>=)  :: t a -> (a -> t b) -> t b
return :: a -> t a

-- Atomic and isolated computations ---
atomic   :: OTM a -> IO a
isolated :: ITM a -> OTM a
retry    :: ITM a
orElse   :: ITM a -> ITM a -> ITM a
```

```
-- Exceptions ------------------------
throw :: Exception e => e -> t a
catch :: Exception e => t a ->
   (e -> t a) -> t a

-- Threading -------------------------
fork :: OTM () -> OTM ThreadId

-- Transactional memory ---------------
data OTVar a
newOTVar   :: a -> ITM (OTVar a)
readOTVar  :: OTVar a -> ITM a
writeOTVar :: OTVar a -> a -> OTM ()
```

Figure 1: The base interface of OTM.

In the rest of this abstract we gradually present the primitives from the OTM library (showed in Figure 1) by some illustrative examples. The discussion is meant to be introductory and informal. The formal semantics has been omitted due to space constraints but it is based on the calculus we presented in [2], with minor variations to accommodate the richer types of OTM. This library has been implemented in Haskell using the standard STM library.

Suppose we want to delegate some long task to another thread and then collect the result once it is ready. An intuitive way to achieve this is by means of *futures*, i.e. "proxy results" that will be produced by the worker threads.

A future can be implemented it OTM as a *transactional variable* `OTVar` holding a value of type `Maybe a` i.e. a type that is "not-ready-yet" (`Nothing`) or actually holds something of type `a` (e.g. "`Just 42`"):

```
type Future a = OTVar (Maybe a)
```

To access the promised value a call to `getFuture f` should block if the value is not ready yet. In OTM (and also STM) blocking a thread translates into "this thread has been scheduled too early" and the scheduler is informed of this fact by means of the primitive `retry`. Therefore we can implement `getFuture` as aside. Note that there is no point in blindly restarting the transaction until at least one of the transactional variables has been modified. Instead an implementation will use the information contained in the transaction log (which is needed by the optimistic execution strategy) to watch for `f` to change. The above snippet is executed in isolation to guarantee consistency between reading and testing the case (actually, it is also an implementation in STM).

```
getFuture::Future a -> ITM a
getFuture f = case (readOTVar f) of
   Nothing -> retry
   Just val -> return val
```

Transaction openness comes into play when a worker is spawn. STM does not allow for thread creation, thus forcing us to implement `spawn` in `IO`. This is indeed possible, but forking and creating the future cannot be guaranteed to be atomic, let alone the creation of several workers. For instance, being able to create workers and futures inside an open transaction allows us to propagate exceptions and abort to all workers. Therefore we can implement `spawn` as follows:

```
spawn :: OTM a -> OTM (Future a)
spawn work = do
      future <- newTVar Nothing
      fork (worker future)
      return future
```

```
worker :: Future a -> OTM ()
worker v = do
   res <- work
   writeOTVar (Just v) $! res
```

Because of its type, spawning multiple computations inside the same transaction is as simple as function composition:

```
spawnMany:: [OTM a] -> OTM [Future a] getAllFutures:: [Future a] -> ITM [a]
spawnMany = mapM spawn                getAllFutures = mapM getFuture
```

While programming with OTM we should prefer the strictest kind of transactions otherwise we will loose some information because of a less precise type. For instance, consider the following alternative implementations of `getAllFutures`:

```
get' = isolated . (mapM getFuture)    get'' = mapM (isolated getFuture)
```

Although both have type `[Future a] -> OTM [a]`, the first is executed in isolation (like `getFutures`) whereas the second allows workers to proceeds during the traversal of the list. Same type, different degree of concurrency.

Transactions can be composed as *alternatives* thanks to the primitive `orElse` which firstly attempts to execute its first argument as a sub-transaction and its second whenever the first one retries. The following function collects the value that is made available first:

```
getAnyFuture (f:fs) = (getFuture f) ‘orElse‘ (getAnyFuture fs)
```

Example: Petri nets Petri nets might be easily implemented in OTM: places are transactional variables holding a number of tokens (`OTvar Peano`). Tokens can be added and removed with the latter operation being blocking.

```
data Place = OTVar Peano              newPlace :: Peano -> ITM Place
                                      newPlace = newOTVar
take :: Place -> ITM ()
take var = do                         put :: Place -> ITM ()
   t <- redOTVar var                  put var = do
   case t of                             v <- readOTVar var
      Zero -> retry                      writeOTVar var (Succ v)
      Succ v -> writeOTVar var v
```

Transitions are `IO` threads that repeatedly consume tokens from their input places and produce tokens to their output places, atomically:

```
transition :: [Place] -> [Place] -> IO ThreadId
transition ins outs = forkIO (forever fire)
   where
      fire :: IO ()
      fire = atomic $ do
         (isolated take) ‘all‘ ins
         (isolated put)  ‘all‘ outs
      all :: (a -> OTM b) -> [a] -> OTM ()
      all f = mapM_ f
```

Although each transition fires sequentially, the firing of different transitions happens in a true concurrent way since transactions are open and isolation is limited to each take/put operation.

References

[1] M. Herlihy and J. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. ISCA*, pages 289–300. ACM, 1993.

[2] M. Miculan, M. Peressotti, and A. Toneguzzo. Open transactions on shared memory. In *Proc. COORDINATION*, pages 213–229, 2015.

[3] N. Shavit, D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.