DOI:10.1145/1378704.1378725

Composable Memory Transactions

By Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy

Abstract

Writing concurrent programs is notoriously difficult and is of increasing practical importance. A particular source of concern is that even correctly implemented concurrency abstractions cannot be composed together to form larger abstractions. In this paper we present a concurrency model, based on *transactional memory*, that offers far richer composition. All the usual benefits of transactional memory are present (e.g., freedom from low-level deadlock), but in addition we describe modular forms of *blocking* and *choice* that were inaccessible in earlier work.

1. INTRODUCTION

The free lunch is over.[25] We have been used to the idea that our programs will go faster when we buy a next-generation processor, but that time has passed. While that next-generation chip will have more CPUs, each individual CPU will be no faster than the previous year's model. If we want our programs to run faster, we must learn to write parallel programs.

Writing parallel programs is notoriously tricky. Mainstream lock-based abstractions are difficult to use and they make it hard to design computer systems that are reliable and scalable. Furthermore, systems built using locks are difficult to compose without knowing about their internals.

To address some of these difficulties, several researchers (including ourselves) have proposed building programming language features over *software transactional memory* (STM), which can perform groups of memory operations atomically.[23] Using transactional memory instead of locks brings well-known advantages: freedom from deadlock and priority inversion, automatic roll-back on exceptions or timeouts, and freedom from the tension between lock granularity and concurrency.

Early work on software transactional memory suffered several shortcomings. Firstly, it did not prevent transactional code from bypassing the STM interface and accessing data directly at the same time as it is being accessed within a transaction. Such conflicts can go undetected and prevent transactions executing atomically. Furthermore, early STM systems did not provide a convincing story for building operations that may block—for example, a shared work-queue supporting operations that wait if the queue becomes empty.

Our work on STM-Haskell set out to address these problems. In particular, our original paper makes the following contributions:

- We re-express the ideas of transactional memory in the setting of the purely functional language Haskell (Section 3). As we show, STM can be expressed particularly elegantly in a declarative language, and we are able to use Haskell's type system to give far stronger guarantees than are conventionally possible. In particular, we guarantee "strong atomicity"[15] in which transactions always appear to execute atomically, no matter what the rest of the program is doing. Furthermore transactions are compositional: small transactions can be glued together to form larger transactions.
- We present a modular form of blocking (Section 3.2). The idea is simple: a transaction calls a `retry` operation to signal that it is not yet ready to run (e.g., it is trying to take data from an empty queue). The programmer does not have to identify the condition which will enable it; this is detected automatically by the STM.
- The `retry` function allows possibly blocking transactions to be composed in *sequence*. Beyond this, we also provide `orElse`, which allows them to be composed as *alternatives*, so that the second is run if the first retries (see Section 3.4). This ability allows threads to wait for many things at once, like the Unix `select` system call—except that `orElse` composes, whereas `select` does not.

Everything we describe is fully implemented in the Glasgow Haskell Compiler (GHC), a fully fledged optimizing compiler for Concurrent Haskell; the STM enhancements were incorporated in the GHC 6.4 release in 2005. Further examples and a programmer-oriented tutorial are also available.[19]

Our main war cry is *compositionality*: a programmer can control atomicity and blocking behavior in a modular way that respects abstraction barriers. In contrast, lock-based approaches lead to a direct conflict between abstraction and concurrency (see Section 2). Taken together, these ideas offer a qualitative improvement in language support for modular concurrency, similar to the improvement in moving from assembly code to a high-level language. Just as with assembly code, a programmer with sufficient time and skills may obtain better performance programming directly with low-level concurrency control mechanisms rather than transactions—but for all but the most demanding applications, our higher-level STM abstractions perform quite well enough.

This paper is an abbreviated and polished version of an earlier paper with the same title.[9] Since then there has been a tremendous amount of activity on various aspects of transactional memory, but almost all of it deals with the question of *atomic memory update*, while much less attention is paid to our central concerns of *blocking* and *synchronization* between threads, exemplified by `retry` and `orElse`. In our view this is a serious omission: locks without condition variables would be of limited use.

Transactional memory has tricky semantics, and the original paper gives a precise, formal semantics for transactions, as well as a description of our implementation. Both are omitted here due to space limitations.

2. BACKGROUND

Throughout this paper we study concurrency between threads running on a shared-memory machine; we do not consider questions of external interaction through storage systems or databases, nor do we address distributed systems. The kinds of problem we have in mind are building collection classes (queues, lists, and so on) and other data structures that concurrent threads use to maintain shared information. There are many other approaches to concurrency that we do not discuss, including data-parallel abstractions from languages like NESL[2] and those from the high-performance computing community such as OpenMP and MPI.

Even in this restricted setting, concurrent programming is extremely difficult. The dominant programming technique is based on *locks*, an approach that is simple and direct, but that simply does not scale with program size and complexity. To ensure *correctness*, programmers must identify which operations conflict; to ensure *liveness*, they must avoid introducing deadlock; to ensure good *performance*, they must balance the granularity at which locking is performed against the costs of fine-grain locking.

Perhaps the most fundamental objection, though, is that *lock-based programs do not compose*. For example, consider a hash table with thread-safe insert and delete operations. Now suppose that we want to delete one item `A` from table `t1`, and insert it into table `t2`; but the intermediate state (in which neither table contains the item) must not be visible to other threads. Unless the implementer of the hash table anticipates this need, there is simply no way to satisfy this requirement without somehow locking out *all* other accesses to the table. One approach is to expose concurrency control methods such as `LockTable` and `UnlockTable`—but this breaks the hash table abstraction, and invites lock-induced deadlock, depending on the order in which the client takes the locks, or race conditions if the client forgets. Yet more complexity is required if the client wants to await the presence of `A` in `t1`—but this blocking behavior must not lock the table (else `A` cannot be inserted). In short, operations that are individually correct (insert, delete) cannot be composed into larger correct operations.

The same phenomenon shows up trying to compose alternative blocking operations. Suppose a procedure `p1` waits for one of two input pipes to have data, using a call to the Unix `select` procedure; and suppose another procedure `p2` does the same thing, on two other pipes. In Unix there is no way to perform a `select` between `p1` and `p2`, a fundamental loss of compositionality. Instead, Unix programmers learn awkward programming techniques to gather up all the file descriptors that must be waited for, perform a single top-level `select`, and then dispatch back to the correct handler. Again, two individually correct abstractions, `p1` and `p2`, cannot be composed into a larger one; instead, they must be ripped apart and awkwardly merged, in direct conflict with the goals of abstraction.

Rather than fixing locks, a more promising and radical alternative is to base concurrency control on *atomic memory transactions*, also known as *transactional memory*. We will show that transactional memory offers a solution to the tension between concurrency and abstraction. For example, with memory transactions we can manipulate the hash table thus:

```
atomic {v := delete(t1, A); insert(t2, A, v)}
```

and to wait for either p1 or p2 we can say

```
atomic {p1 `orElse` p2}
```

These simple constructions require no knowledge of the implementation of `insert`, `delete`, `p1` or `p2`, and they continue to work correctly if these operations may block, as we shall see.

2.1. Transactional memory

The idea of transactions is not new. They have been a fundamental mechanism in database design for many years, and there has been much subsequent work on transactional memory. Larus and Rajwar provide a recent survey.[14]

The key idea is that a block of code, including nested calls, can be enclosed by an `atomic` block, with the guarantee that it runs atomically with respect to every other atomic block. Transactional memory can be implemented using *optimistic synchronization*. Instead of taking locks, an `atomic` block runs without locking, accumulating a thread-local *transaction log* that records every memory read and write it makes. When the block completes, it first *validates* its log, to check that it has seen a consistent view of memory, and then *commits* its changes to memory. If validation fails, because memory read by the method was altered by another thread during the block's execution, then the block is re-executed from scratch.

Suitably implemented transactional memory eliminates many of the low-level difficulties that plague lock-based programming. There are no lock-induced deadlocks (because there are no locks); there is no priority inversion; and there is no painful tension between granularity and concurrency. However, initial work made little progress on transactional abstractions that compose well. There are three particular problems.

Firstly, since a transaction may be rerun automatically, it is essential that it does nothing irrevocable. For example, the transaction

```
atomic {if (n > k) then launchMissiles(); S2}
```

might launch a second salvo of missiles if it were re-executed. It might also launch the missiles inadvertently if, say, the thread was de-scheduled after reading `n` but before reading `k`, and another thread modified both before the thread was resumed. This problem begs for a guarantee that the body of the `atomic` block can only perform memory operations, and hence can only make benign modifications to its own transaction log, rather than performing irrevocable input/output.

Secondly, many systems do not support synchronization *between* transactions, and those that do rely on a a programmer-supplied Boolean guard on the `atomic` block.[8] For example, a method to get an item from a buffer might be:

```
Item get () {
  atomic (n_items > 0) {... remove item ...}
}
```

The thread waits until the guard (`n_items > 0`) holds, before executing the block. But how could we take two *consecutive* items? We cannot call `get();get()`, because another thread might perform an intervening `get`. We could try wrapping two calls to `get` in a nested `atomic` block, but the semantics of this are unclear unless the outer block checks there are two items in the buffer. This is a disaster for abstraction, because the client (who wants to get the two items) has to know about the internal details of the implementation. If several separate abstractions are involved, matters are even worse.

Thirdly, no previous transactional memory supports *choice*, exemplified by the `select` example mentioned earlier.

We tackle all three issues by presenting transactional memory in the context of the declarative language Concurrent Haskell, which we briefly review next.

2.2. Concurrent Haskell

Concurrent Haskell[20] is an extension to Haskell 98, a pure, lazy, functional language. It provides explicitly forked threads, and abstractions for communicating between them. This naturally involves side effects and so, given the lazy evaluation strategy, it is necessary to be able to control exactly when they occur. The big breakthrough came from a mechanism called *monads*.[21]

Here is the key idea: a value of type `IO a` is an *I/O action* that, when performed, may do some I/O before yielding a value of type `a`. For example, the functions `putChar` and `getChar` have types:

```
putChar :: Char -> IO ()
getChar :: IO Char
```

That is, `putChar` takes a `Char` and delivers an I/O action that, when performed, prints the character on the standard output; while `getChar` is an action that, when performed, reads a character from the console and delivers it as the result of the action. A complete program must define an I/O action called `main`; executing the program means performing that action. For example:

```
main :: IO ()
main = putChar 'x'
```

I/O actions can be glued together by a *monadic bind* combinator. This is normally used through some syntactic sugar, allowing a C-like syntax. Here, for example, is a complete program that reads a character and then prints it twice:

```
main = do {c <- getChar; putChar c; putChar c}
```

As well as performing external input/output, I/O actions include operations with side effects on *mutable cells*. A value of type `IORef a` is a mutable storage cell which can hold values of type `a`, and is manipulated (only) through the following interface:

```
newIORef   :: a -> IO (IORef a)
readIORef  :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

`newIORef` takes a value of type `a` and creates a mutable storage location holding that value. `readIORef` takes a reference to such a location and returns the value that it contains. `writeIORef` provides the corresponding update operation. Since these cells can only be created, read, and written using operations in the `IO` monad, there is a type-secure guarantee that ordinary functions are unaffected by state—for example, a pure function `sin` cannot read or write an `IORef` because `sin` has type `Float -> Float`.

Concurrent Haskell supports threads, each independently performing input/output. Threads are created using a function `forkIO`:

```
forkIO :: IO a -> IO ThreadId
```

`forkIO` takes an I/O action as its argument, spawns a fresh thread to perform that action, and immediately returns its thread identifier to the caller. For example, here is a program that forks a thread that prints 'x', while the main thread goes on to print 'y':

```
main = do {forkIO (print 'x'); print 'y'}
```

Peyton Jones provides a fuller introduction to concurrency, I/O, exceptions and cross-language interfacing (the "awkward squad" for pure, lazy, functional programming),[18] and Daume III provides a general online tutorial to Haskell.[6]

3. COMPOSABLE TRANSACTIONS

We are now ready to present the key ideas of the paper. Our starting point is this: *a purely declarative language is a perfect setting for transactional memory*, for two reasons. First, the type system explicitly separates computations which may have side effects from effect-free ones. As we shall see, it is easy to refine it so that transactions can perform memory effects but not irrevocable input/output effects. Second, reads from and writes to mutable cells are explicit, and relatively rare: most computation takes place in the purely functional world. These functional computations perform many, many memory operations—allocation, update of thunks, stack operations, and so on—but none of these need to be tracked by the STM, because they are pure and never need to be rolled back. Only the relatively rare explicit operations need be logged, so a software implementation is entirely appropriate.

So our approach is to use Haskell as a kind of "laboratory" in which to study the ideas of transactional memory in a setting with a very expressive type system. As we go, we will mention primitives from the `STM` library, whose interface is summarized in Figure 1. In this paper, we focus on examples of how STM can be used in building simple concurrency abstractions. Our original paper[9] formally defines the details of the design via an operational semantics which we developed alongside our implementations; we found this invaluable in highlighting interactions between the constructs—for example, what happens if an exception is raised deep inside an atomic block, nested within catch handlers and `orElse`? For the moment we return to simpler examples.

Figure 1: The STM interface.

```
-- The STM monad itself
data STM a
instance Monad STM
-- Monads support "do" notation and sequencing

-- Exceptions
throw  ::  Exception -> STM a
catch  ::  STM a -> (Exception->STM a) -> STM a

-- Running STM computations
atomic  ::  STM a -> IO a
retry   ::  STM a
orElse  ::  STM a -> STM a -> STM a

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

3.1. Transactional variables and atomicity

Suppose we wish to implement a resource manager, which holds an integer-valued resource. The call `(getR r n)` should acquire `n` units of resource `r`, blocking if `r` holds insufficient resource; the call `(putR r n)` should return `n` units of resource to `r`.

Here is how we might program `putR` in STM-Haskell:

```
type Resource = TVar Int
putR :: Resource -> Int -> STM ()
putR r i = do { v <- readTVar r
              ; writeTVar r (v + i)}
```

The currently available resource is held in a *transactional variable* of type `TVar  Int`. The `type` declaration simply gives a name to this type. The function `putR` reads the value `v` of the resource from its cell, and writes back `(v + i)` into the same cell. (We discuss `getR` next, in Section 3.2.)

The `readTVar` and `writeTVar` operations both return STM actions (Figure 1), but Haskell allows us to use the same `do {...}` syntax to compose STM actions as we did for I/O actions. These STM actions remain tentative during their execution: to expose an STM action to the rest of the system, it can be passed to a new function `atomic`, with type:

```
atomic :: STM a -> IO a
```

It takes a memory transaction, of type `STM  a`, and delivers an I/O action that, when performed, runs the transaction atomically with respect to all other memory transactions. One might say:

```
main = do {...; atomic (putR r 3); ...}
```

The underlying transactional memory deals with maintaining a per-thread transaction log to record the tentative accesses made to `TVar`s. When `atomic` is invoked, the STM checks that the logged accesses are *valid*—i.e., no concurrent transaction has committed conflicting updates to those `TVar`s. If the log is valid then the STM *commits* it atomically to the heap, thereby exposing its effects to other transactions. Otherwise the memory transaction is rerun with a fresh log.

Splitting the world into STM actions and I/O actions provides two valuable properties, both statically checked by the type system:

- There is no way to perform general I/O within a transaction, because there is no operation that takes an IO computation and performs it in the STM monad. Hence only STM actions and pure computation can be performed inside a memory transaction. This is precisely the guarantee we sought in Section 2.1. It statically prevents the programmer from calling `launchMissiles` inside a transaction, because launching missiles is an I/O action with type `IO ()`, and cannot be composed with STM actions.
- No STM actions can be performed outside a transaction, so the programmer cannot accidentally read or write a `TVar` without the protection of `atomic`. Of course, one can always say `atomic (readTVar v)` to read a `TVar` in a trivial transaction, but the call to `atomic` cannot be omitted.

3.2. Blocking memory transactions

Any concurrency mechanism must provide a way for a thread to await an event or events caused by other threads. In lock-based programming, this is typically done using condition variables; message-based systems offer a construct to wait for messages on a number of channels; POSIX provides `select`; Win32 provides `WaitForMultipleObjects`; and STM systems to date allow the programmer to guard the atomic block with a Boolean condition (see Section 2.1).

The Haskell setting led us to a remarkably simple new mechanism for blocking. Furthermore, as we show in Sections 3.3 and 3.4, it supports composition in ways that are not possible with lock-based programming.

The idea is to provide a retry operation to indicate that the current atomic action is not yet ready to run to completion. Here is the code for getR:

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

It reads the value v of the resource and, if v >= i, decreases it by i. If v < i, there is insufficient resource in the variable, in which case it calls retry. Conceptually, retry aborts the transaction with no effect, and restarts it at the beginning. However, there is no point in actually re-executing the transaction until *at least one of the TVars read during the attempted transaction has been written by another thread*. Happily, the transaction log (which is needed anyway) already records exactly which TVars were read. The implementation, therefore, blocks the thread until at least one of these is updated. Notice that retry's type (STM a) allows it to be used wherever an STM action may occur.

Unlike the validation check, which is automatic and implicit, retry is called explicitly by the programmer. It does not indicate anything bad or unexpected; rather, it shows up when some kind of blocking would take place in other approaches to concurrency.

Notice that there is no need for the putR operation to remember to signal any condition variables. Simply by writing to the TVars involved, the producer will wake up the consumer. A whole class of lost-wake-up bugs is thereby eliminated.

From an efficiency point of view, it makes sense to call retry as early as possible, and to refrain from reading unrelated locations until after the test succeeds. Nevertheless, the programming interface is delightfully simple, and easy to reason about.

3.3. Sequential composition

By using atomic, the programmer identifies atomic transactions, in the classic sense that the entire set of operations that it contains appears to take place indivisibly. This is the key to sequential composition for concurrency abstractions. For example, to grab three units of one resource and seven of another, a thread can say

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

The standard do {..;..} notation combines the STM actions from the two getR calls and the underlying transactional memory commits their updates as a single atomic I/O action.

The retry function is central to making transactions composable when they may block. The transaction above will block if either r1 or r2 has insufficient resource: there is no need for the caller to know how getR is implemented, or what condition guarantees its success. Nor is there any risk of deadlock by awaiting r2 while holding r1.

This ability to compose STM actions is why we did not define getR as an I/O action, wrapped in a call to atomic. By leaving it as an STM action, we allow the programmer to compose it with other STM actions before finally sealing it into a transaction with atomic. In a lock-based setting, one would worry about crucial locks being released between the two calls, and about deadlock if another thread grabbed the resources in the opposite order, but there are no such concerns here.

The STM type on an atomic action provides a strong guarantee: the *only* way the action can be executed is for it to be passed to atomic. *Any STM action can be robustly composed with other STM actions: the resulting sequence will still execute atomically.*

3.4. Composing alternatives

We have discussed composing transactions in *sequence*, so that both are executed. STM-Haskell also allows us to compose transactions as *alternatives*, so that only one is executed. For example, to get *either* 3 units from r1 *or* 7 units from r2:

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

The orElse function is provided by the STM module (Figure 1); here, it is written infix, by enclosing it in backquotes, but it is a perfectly ordinary function of two arguments. The transaction s1 `orElse` s2 first runs s1; if s1 calls retry, then s1 is abandoned with no effect, and s2 is run. If s2 also calls retry then the entire call retries—but it waits on the variables read by *either* of the two nested transactions (i.e., on the union of two variable sets). Again, the programmer needs know nothing about the enabling conditions of s1 and s2.

Using orElse provides an elegant way for library implementers to defer to their caller the question of whether or not to block. For instance, it is straightforward to convert the blocking version of getR into one which returns a Boolean success or failure result:

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

If getR completes normally, nonBlockGetR will return True; on the other hand, if getR blocks (i.e., retries), the orElse will try its second alternative, which succeeds immediately, returning False. Notice that this idiom depends on the left-biased nature of orElse. The same kind of construction can be also used to build a blocking operation from one that returns a Boolean result: simply invoke retry on receiving a False result:

```
blockGetR :: Resource -> Int -> STM ()
blockGetR r i =
        do {s <- nonBlockGetR r i;
            if s then return () else retry}
```

The `orElse` function obeys useful laws: it is associative and has unit `retry`:

```
M1 ‘orElse‘ (M2 ‘orElse‘ M3)
       = (M1 ‘orElse‘ M2) ‘orElse‘ M3
retry ‘orElse‘ M = M
M ‘orElse‘ retry = M
```

Haskell aficionados will recognize that STM may thus be an instance of MonadPlus.

3.5. Exceptions

The `STM` monad supports exceptions just like the `IO` monad, and in much the same way as (say) C#. Two new primitive functions, `catch` and `throw`, are required; their types are given in Figure 1. The question is: how should transactions and exceptions interact? For example, what should this transaction do?

```
atomic (do
   { n <- readTVar v_n
   ; lim <- readTVar v_lim
   ; writeTVar v_n (n + 1)
   ; if n > lim
     then throw
               (AssertionFailed ”Urk”)
     else if (n == lim) then retry
     else return ()
   ; ... write data into buffer...})
```

The programmer throws an exception if `n > lim`, in which case the...`write data`... part will clearly not take place. But what about the write to `v_n` from before the exception was thrown?

Concurrent Haskell encourages programmers to use exceptions for signalling error conditions, rather than for normal control flow. Built-in exceptions, such as divide-by-zero, also fall into this category. For consistency, then, in the above program *we do not want the programmer to have to take account of the possibility of exceptions*, when reasoning that if `v_n` is (observably) written then data is written into the buffer. We, therefore, specify that exceptions have *abort semantics*: if an atomic transaction throws an exception, then the transaction must be validated as if it had completed normally; however, no changes are committed. If validation succeeds, then the exception is propagated; but if validation fails, then the throwing of the exception may have been based on an inconsistent view of memory, so the exception is discarded and the transaction is re-executed from scratch. Abort semantics make it much easier to reason about invariants: the programmer only has to worry about the invariant being preserved when the transaction commits; exceptions raised during the transaction always restore the invariant, by definition.

Our use of exceptions to abort `atomic` blocks is a free design choice. In other languages, especially in ones where exceptions are used more frequently, it might be appropriate to distinguish exceptions that cause the enclosing `atomic` block to abort from exceptions that allow it to commit before they are propagated.

Notice the difference between calling `throw` and calling `retry`. The former signals an error, and aborts the transaction; the latter only indicates that the transaction is not yet ready to run, and causes it to be re-executed when the situation changes.

An exception can carry a value out of the STM world. For example, consider

```
atomic (do
   { s <- readTVar svar
   ; writeTVar svar ”Wuggle”
   ; is length s < 10 then
         throw (AssertionFailed s)
     else ...})
```

Here, the external world gets to see the exception value holding the string `s` that was read out of the `TVar`. However, since the transaction is aborted before the exception propagates, its write to `svar` is not externally observable. One might argue that it is wrong to allow even reads to “leak” from an aborted transaction, but we do not agree. The values carried by an exception can only represent a consistent view of the heap (or validation would fail, and the transaction would re-execute without propagating the exception), and it is almost impossible to debug an error condition that only says “something bad happened” while deliberately discarding all clues to what the bad thing was. The basic transactional guarantees are not threatened.

What if the exception carries a `TVar` *allocated* in the aborted transaction? A dangling pointer would be unpleasant. To avoid this we refine the semantics of exceptions to say that a transaction that throws an exception is aborted so far as its write effects are concerned, but its *allocation* effects are retained; after all, they are thread-local. As a result, the `TVar` is visible after the transaction, in the state it had when it was allocated. Cases like these are tricky, which is why we developed a full formal semantics.[9]

Concurrent Haskell also provides asynchronous exceptions which can be thrown into a thread as a signal—typical examples are error conditions like stack overflow, or when a master thread wishes to shut down a helper. If a thread is in the midst of an STM transaction, then the transaction log can be discarded without externally visible effects.

What if an exception is raised inside `orElse`? We considered a design in which, if the first alternative throws an exception, we could discard its effects and try the second alternative instead. But that would invalidate the beautiful identify which makes `retry` a unit for `orElse` and would also make `orElse` asymmetric in its treatment of exceptions (discarded from the first alternative but propagated by the second). We, therefore, chose that exceptions *do* propagate from the first alternative: the second alternative is examined only if the first one calls a `retry`.

What about catching an exception within an `atomic` block? Consider this example:

```
f :: Port Int -> STM ()
f p = do { item <- readPort p
         ; g item}
```

If g goes wrong (throws an exception), the author of f might reasonably want to ensure that the item is not read from the port p and then discarded. And indeed, if f is called in an atomic context, such as atomic (f p), the effects of readPort are discarded, so that the item is not read. But suppose f is called in a context that catches the exception *before leaving the STM world*:

```
bad :: Port Int -> Port Int -> STM ()
bad p1 p2 = catch (f p1) (\exn -> f p2)
```

In our original paper we proposed that the effects of (f p1) would be retained and be visible to the call (f p2). Furthermore, if the latter succeeds without itself throwing an exception or retrying, the effects of (f p1) would be permanently committed.

Ultimately we felt that this treatment of effects that precede an exception seemed inconsistent. Consider the author of f; in an effort to ensure that the item is indeed not read if g throws an exception, he might try this:

```
f :: Port Int -> STM ()
f p = do { item <- readPort p
         ; catch (g item)
                 (recover exn item)}
    where
     recover exn item
       = do { unReadPort p item
            ; throw exn}
```

But that relies on the existence of unReadPort to manually replicate the roll-back supported by the underlying STM. The conclusion is clear: the effects of the first argument of catch should be reverted if the computation raises an exception. Again, this works out nicely in the context of STM-Haskell because the catch operation used here has an STM type, which indicates to the programmer that the code is transactional.

4. APPLICATIONS AND EXAMPLES

In this section we provide some examples of how composable memory transactions can be used to build higher-level concurrency abstractions. We focus on operations that involve potentially blocking communication between threads. Previous work has shown, many times over, how standard shared-memory data structures can be developed from sequential code using transactional memory operations.[8,11]

4.1. MVars

Prior to our STM work, Concurrent Haskell provided MVars as its primitive mechanism for allowing threads to communicate safely. An MVar is a mutable location like a TVar, except that it may be either *empty*, or *full* with a value. The take MVar function leaves a full MVar empty, but blocks on an empty MVar. A putMVar on an empty MVar leaves it full, but blocks on a full MVar. So MVars are, in effect, a one-place channel.

It is easy to implement MVars on top of TVars. An MVar holding a value of type a can be represented by a TVar holding a value of type Maybe a; this is a type that is either an empty value ("Nothing"), or actually holds an a (e.g., "Just 42").

```
type MVar a = TVar (Maybe a)
newEmptyMVar :: STM (MVar a)
newEmptyMVar = newTVar Nothing
```

The takeMVar operation reads the contents of the TVar and retries until it sees a value other than Nothing:

```
takeMVar :: MVar a -> STM a
takeMVar mv
  = do { v <- readTVar mv
       ; case v of
           Nothing  -> retry
           Just val -> do { writeTVar mv Nothing
                          ; return val}}
```

The corresponding putMVar operation retries until it sees Nothing, at which point it updates the underlying TVar:

```
putMVar :: MVar a -> a -> STM ()
putMVar mv newval
  = do { v <- readTVar mv
       ; case v of
           Nothing  -> writeTVar mv
                           (Just newval)
           Just val -> retry}
```

Notice how operations that return a Boolean success / failure result can be built directly from these blocking designs. For instance:

```
tryPutMVar :: MVar a -> a -> STM Bool
tryPutMVar mv val
   = do {putMVar mv val ; return True}
     `orElse` return False
```

4.2. Multicast channels

MVars effectively provide communication channels with a single buffered item. In this section we show how to program buffered, multi-item, multicast channels, in which items written to the channel (writeMChan in the interface below) are buffered internally and received once by each read-port created from the channel. The full interface is:

```
data MChan a
data port a
newMChan   :: STM (MChan a)
-- Write an item to the channel:
writeMChan :: MChan a -> a -> STM ()
-- Create a new read port:
newPort    :: MChan a -> STM (Port a)
-- Read the next buffered item:
readPort   :: Port a -> STM a
```

We represent the buffered data by a linked list, or `Chain`, of items, with a transactional variable in each tail, so that it can be extended by `writeMChan`:

```
type Chain a = TVar (Item a)
data Item a = Empty | Full a (Chain a)
```

An `MChan` is represented by a mutable pointer to the "write" end of the chain, while a `Port` points to the read end:

```
type MChan a = TVar (Chain a)
type Port a = TVar (Chain a)
```

With these definitions, the code writes itself:

```
newMChan = do {c <- newTVar Empty; newTVar c}
newPort mc = do {c <- readTVar mc; newTVar c}

readPort p
  = do { c <- readTVar p
       ; i <- readTVar c
       ; case i of
           Empty     -> retry
           Full v c' -> do {writeTVar p c';
                            return v}}

writeMChan mc v
= do { c <- readTVar mc
     ; c' <- newTVar Empty
     ; writeTVar c (Full v c')
     ; writeTVar mc c'}
```

Notice the use of `retry` to block `readPort` when the buffer is empty. Although this implementation is very simple, it ensures that each item written into the `MChan` is delivered to every `Port`; it allows multiple writers (their writes are interleaved); it allows multiple readers on each port (data read by one is not seen by the other readers on that port); and when a port is discarded, the garbage collector recovers the buffered data.

More complicated variants are simple to program. For example, suppose we wanted to ensure that the writer could get no more than *N* items ahead of the most advanced reader. One way to do this would be for the writer to include a serially increasing `Int` in each `Item`, and have a shared `TVar` holding the maximum serial number read so far by any reader. It is simple for the readers to keep this up to date, and for the writer to consult it before adding another item.

4.3. Merge

We have already stressed that transactions are *composable*. For example, to read from either of the two different multicast channels, we can say:

```
atomic (readPort p1 `orElse` readPort p2)
```

No changes need to be made to either multicast channel. If neither port has any data, the STM machinery will cause the thread to wait simultaneously on the `TVars` at the extremity of each channel.

Equally, the programmer can wait on a condition that involves a mixture of `MVars` and `MChans` (perhaps the multicast channel indicates ordinary data and an `MVar` is being used to signal a termination request), for instance:

```
atomic (readPort p1 `orElse` takeMVar m1)
```

This example is contrived for brevity, but it shows how operations taken from different libraries, implemented without anticipation of their being used together, can be composed. In the most general case, we can select between values received from a number of different sources. Given a list of computations of type `STM a` we can take the first value to be produced from any of them by defining a merge operator:

```
merge  ::  [STM a] -> STM a
merge  =   foldr1 orElse
```

(The function `foldr1 f` simply reduces a list $[a_1\, a_2 \cdots a_n]$ to the value a_1 `` `f` `` a_2 `` `f` `` $\cdots$ `` `f` `` a_n.) This example is childishly simple in STM-Haskell. In contrast, a function of type

```
mergeIO :: [IO a] -> IO a
```

is unimplementable in Concurrent Haskell, or indeed in other settings with operations built from mutual exclusion locks and condition variables.

5. IMPLEMENTATION

Since our original paper there has been a lot of work on building fast implementations of STM along with hardware support to replace or accelerate them.[14] The techniques we have used in STM-Haskell are broadly typical of much of this work and so we do not go into the details here. In summary, however, while a transaction is running, it builds up a private log that records the `TVars` it has accessed, the values it has read from them and (in the case of writes) the new values that it wants to store to them. When a transaction attempts to commit, it has to reconcile this log with the heap. Logically this has two steps: *validating* the transaction to check that there have been no conflicting updates to the locations read, and then *writing-back* the updates to the `TVar`s that have been modified.

However, the `retry` and `orElse` abstractions led us to think more carefully about how to integrate blocking operations with this general approach. Following Harris and Fraser's work[8] we built `retry` by using a transaction's log to identify the `TVar`s that it has read and then adding "trip wires" to those `TVar`s before blocking: subsequent updates to any of those `TVar`s will unblock the thread.

The `orElse` and `catch` constructs are both implemented using closed nested transactions[17] so that the updates made by the enclosed work can be rolled back without discarding the outer transaction. There is one subtlety that we did not appreciate in our original paper: if the enclosed transaction is rolled back *then the log of locations it has read must be retained by the parent*. In retrospect the reason is clear—the decision of whether or not to roll back must be validated at the same atomic point as the outer transaction.

5.1. Progress

The STM implementation guarantees that one transaction can force another to abort only when the first one commits. As a result, the STM implementation is *lock-free* in the sense that it guarantees at any time that some running transaction can successfully commit. For example, no deadlock will occur if one transaction reads and writes to `TVar` x and then `TVar` y, while a second reads and writes to those TVars in the opposite order. Each transaction will observe the original value of those `TVar`s; the first to validate will commit, and the second will abort and restart. Similarly, synchronization conflicts over `TVar`s cannot cause cyclic restart, where two or more transactions repeatedly abort one another.

Starvation is possible, however. For example, a transaction that runs for a very long time may repeatedly be aborted by shorter transactions that conflict with it. We think that starvation is unlikely to occur in practice, but we cannot tell without further experience. A transaction may also never commit if it is waiting for a condition that never becomes true.

6. RELATED WORK

Transactions have long been used for fault tolerance in databases[7] and distributed systems. These transactions rely on stable storage and distributed commit protocols to protect system integrity against crashes and communication failures. Transactional memory of the kind we are studying provides access to memory within a single process; it is not intended to survive crashes, so there is no need for distributed commit protocols or stable storage. It follows that many design and implementation issues are quite different from those arising in distributed or persistence-only transaction systems. TM was originally proposed as a hardware architecture[12,24] to support nonblocking synchronization, and architectural support for this model remains the subject of ongoing research, as does the construction of efficient implementations in software. Larus and Rajwar provide a recent survey of implementation techniques.[14]

Transactional composition requires the ability to run transactions of arbitrary size and duration, presenting a challenge to hardware-based transactional memory designs, which are inherently resource-limited. One way for hardware to support large transactions is by virtualization,[4,22] providing transparent overflow mechanisms. Another way is by hybrid STM designs[5,13] that combine both hardware and software mechanisms.

After our original paper, Carlstrom et al. examined a form of `retry` that watches for updates to a specified set of locations,[3] arguing that this is easier to support in hardware and may be more efficient than our form of `retry`. However, unless the watch set is defined carefully, this sacrifices the composability that `retry` provides because updates to non-watched locations may change the control flow within the transaction.

Our original paper also discusses related programming abstractions for concurrency, notably Concurrent ML's *composable events* and Scheme48's *proposals*.

7. CONCLUSION

In this paper we have introduced the ideas from STM-Haskell for composable memory transactions, providing a substrate for concurrent programming that offers far richer composition than has been available to date: two atomic actions can be glued together in sequence with the guarantee that the result will run atomically, and two atomic actions can be glued together as alternatives with the guarantee that exactly one of them will run. In subsequent work we have further enhanced the STM interface with *invariants*.[10]

We have used Haskell as a particularly suitable laboratory to explore these ideas and their implementation. An obvious question is this: to what extent can our results be carried back into the mainstream world of imperative programming? This is a question that we and many others have been investigating since our original paper. The ideas of composable blocking through `retry` and `orElse` seem straightforward to apply in other settings—subject, of course, to support for blocking and wake-up within the lower levels of the systems.

A more subtle question is the way in which our separation between transacted state and nontransacted state can be applied, or our separation between transacted code and nontransacted code. In Haskell, mutable state and impure code are expected to be the exception rather than the norm, and so it seems reasonable to distinguish the small amount of impure transacted code from the small amount of impure nontransacted code; both, in any case, can call into pure functions.

In contrast, in mainstream languages, most code is written in an impure style using mutable state. This creates a tension: statically separating transacted code and data retains the strong guarantees of STM-Haskell (no irrevocable calls to "`launchMissiles`" within a transaction, and no direct access to transacted state without going through the STM interface), but it requires source code duplication to create transacted variants of library functions and marshaling between transacted data formats and normal data formats.

Investigating the complex trade-offs in this design space is the subject of current research.[1,16]

Whether or not one believes in transactions, it does seem likely that some combination of effect systems and/or ownership types will play an increasingly important role in concurrent programming languages, and these may contribute to the guarantees desirable for memory transactions.

Our main claim is that transactional memory qualitatively raises the level of abstraction offered to programmers. Just as high-level languages free programmers from worrying about register allocation, so transactional memory frees the programmer from concerns about locks and lock acquisition order in designing shared-memory data structures. More fundamentally, one can combine such abstractions without knowing their implementations, a property that is the key to constructing large programs.

Like high-level languages, transactional memory does not banish bugs altogether; for example, two threads can easily deadlock if each awaits some communication from the other. But the gain is very substantial: transactions provide a programming platform for concurrency that eliminates whole classes of concurrency errors, and allows the programmer to concentrate on the really interesting bits.

Acknowledgments

We would like to thank Byron Cook, Austin Donnelly, Matthew Flatt, Jim Gray, Dan Grossman, Andres Löh, Jon Howell, Jan-Willem Maessen, Jayadev Misra, Norman Ramsey, Michael Ringenburg, David Tarditi, and especially Tony Hoare, for their helpful feedback on earlier versions of this paper, and Guy Steele for his meticulous suggestions in preparing this revised version.

References

1. Abadi, M., Birrell, A., Harris, T., and Isard, M. Semantics of transactional memory and automatic mutual exclusion. *POPL'08: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 63–74, ACM, Jan. 2008.
2. Blelloch, G.E., Hardwick, J.C., Sipelstein, J., Zagha, M., and Chatterjee, S. Implementation of a portable nested data-parallel language. J. *Parallel Distrib. Comput.*, 21 (1): 4–14, 1994.
3. Carlstrom, B.D., McDonald, A., Chafi, H., Chung, J., Minh, C.C., Kozyrakis, C., and Olukotun, K. The Atomos transactional programming language. *PLDI'06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1–13, ACM, June 2006.
4. Chung, J., Minh, C.C., McDonald, A., Skare, T., Chafi, H., Carlstrom, B.D., Kozyrakis, C., and Olukotun, K. Tradeoffs in transactional memory virtualization. *ASPLOS'06: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 371–381, ACM, Oct. 2006.
5. Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., and Nussbaum, D. Hybrid transactional memory. *ASPLOS'06: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 336–346, ACM, Oct. 2006.
6. Daume III, H. Yet another Haskell tutorial. http://www.cs.utah.edu/~hal/docs/daume02yaht.pdf, 2006.
7. Gray, J., and Reuter, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1992.
8. Harris, T., and Fraser, K. Language support for lightweight transactions. *OOPSLA'03: Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 388–402, ACM, Oct. 2003.
9. Harris, T., Marlow, S., Peyton Jones, S., and Herlihy, M. Composable memory transactions. *PPoPP'05: Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 48–60, ACM, June 2005.
10. Harris, T., and Peyton Jones, S. Transactional memory with data invariants. *TRANSACT'06: Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.
11. Herlihy, M., Luchangco, V., Moir, M., and Scherer, III, W.N. Software transactional memory for dynamic-sized data structures. *PODC'03: Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, pp. 92–101, ACM, July 2003.
12. Herlihy, M. and Moss, J.E.B. Transactional memory: Architectural support for lock-free data structures. *ISCA'93: Proceedings of the 20th International Symposium on Computer Architecture*, pp. 289–300, ACM, May 1993.
13. Kumar, S., Chu, M., J. Hughes, C., Kundu, P., and Nguyen, A. Hybrid transactional memory. *PPoPP'06: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 209–220, ACM, Mar 2006.
14. Larus, J., and Rajwar, R. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, 2007.
15. Martin, M., Blundell, C., and Lewis, E. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.* 5(2):17, 2006.
16. Moore, K.F. and Grossman, D. High-level small-step operational semantics for transactions. *POPL'08: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 51–62, ACM, Jan. 2008.
17. Moss, E.B. Nested transactions: An approach to reliable distributed computing. Tech. Rep. MIT/LCS/TR-260, Massachusetts Institute of Technology, Apr. 1981.
18. Peyton Jones, S. Tackling the awkward squad: Monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. *Engineering Theories of Software Construction, Marktoberdorf Summer School 2000*.
19. Peyton Jones, S. Beautiful concurrency. In *Beautiful Code* (2007), A. Oran and G. Wilson, Eds., O'Reilly.
20. Peyton Jones, S., Gordon, A., and Finne, S. Concurrent Haskell. *POPL'96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 295–308, ACM, Jan. 1996.
21. Peyton Jones, S. and Wadler, P. Imperative functional programming. *POPL'93: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 71–84, ACM, Jan. 1993.
22. Rajwar, R., Herlihy, M., and Lai, K. Virtualizing transactional memory. *ISCA'05: Proceedings of the 32nd International Symposium on Computer Architecture*, pp. 494–505, IEEE Computer Society, June 2005.
23. Shavit, N., and Touitou, D. Software transactional memory. *PODC'95: Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pp. 204–213, ACM, Aug. 1995.
24. Stone, J.M., Stone, H.S., Heidelberger, P., and Turek, J. Multiple reservations and the Oklahoma update. *IEEE Parallel and Distributed Technology* 1(4):58–71, 1993.
25. Sutter, H. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's J.* (March 2005).

Tim Harris (tharris@microsoft.com) Microsoft Research

Simon Marlow (simonmar@microsoft.com) Microsoft Research

Simon Peyton Jones (simonpj@microsoft.com) Microsoft Research

Maurice Herlihy (mph@cs.brown.edu) Brown University