Composable Memory Transactions

Tim Harris Simon Marlow Simon Peyton Jones Maurice Herlihy

Microsoft Research
7 J J Thomson Avenue, Cambridge, UK, CB3 0FB
{tharris,simonmar,simonpj,t-maherl}@microsoft.com

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 new concurrency model, based on *transactional memory*, that offers far richer composition. All the usual benefits of transactional memory are present (e.g. freedom from deadlock), but in addition we describe new modular forms of *blocking* and *choice* that have been inaccessible in earlier work.

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent Programming – Parallel programming

General Terms: Algorithms, Languages

Keywords: Non-blocking algorithms, locks, transactions

1. INTRODUCTION

Concurrent programming is notoriously tricky. Current lock-based abstractions are difficult to use and 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 *software transactional memory* (STM), which can perform groups of memory operations atomically [27]. 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.

Although promising, our previous work on transactional memory suffered a number of shortcomings: it could not statically prevent threads from bypassing transactional interfaces and it did not provide a convincing story for operations that may block. In this paper we resolve these shortcomings. In particular, we make the following contributions:

- We re-express the ideas of transactional memory in the setting of Concurrent Haskell (Section 3). This is much more than a routine "port" into a new setting. 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. Furthermore transactions are compositional: small transactions can be glued together to form larger transactions.

- We present a new, modular form of blocking, which appears to the programmer as a simple function called `retry` (Section 3.2). Unlike most existing approaches, the programmer does not have to identify the condition under which the transaction can run to completion: `retry` can occur anywhere within the transaction, blocking it until an alternative execution path becomes possible.

- 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 (Section 3.4). This ability allows threads to wait for many things at once, like the Unix `select` system call – except that `orElse` composes well, whereas `select` does not. It turns out that `orElse` requires the underlying STM implementation to support genuine *nested transactions*, the first STM to do so (Section 6.4).

- Unusually for a practical programming language, we provide a formal operational semantics of our system in Section 5. This semantics clarifies the behaviour in cases which have a less intuitive meaning, such as what happens if an exception is raised mid-way through a memory transaction.

- We have implemented our design in the Glasgow Haskell Compiler, a fully-fledged optimising compiler for Concurrent Haskell. The changes are localised, rather than pervasive, and we describe the details in Section 6.

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. Our main war-cry is *compositionality*: a programmer can control atomicity and blocking behaviour in a modular way that respects abstraction barriers. In contrast, current lock-based approaches lead to a direct conflict between abstraction and concurrency (Section 2).

PPoPP'05, June 15–17, 2005, Chicago, Illinois, USA.

2. BACKGROUND

Throughout this paper we study *internal concurrency* between the threads interacting through memory in a single process; we do not consider here the questions of external interaction through storage systems or databases, nor do we address distributed systems.

Even in this 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*: correct fragments may fail when combined.

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 implementor of the hash table anticipates this need, there is simply no way to satisfy this requirement. Even if she does, all she can do is expose methods such as `LockTable` and `UnlockTable` – but as well as breaking the hash-table abstraction, they invite 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 behaviour 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 an internal call to the Unix `select` procedure; and suppose another procedure `p2` does the same thing, on two different 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 recent work on transactional memories [11, 10, 9, 6, 31].

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 synchronisation*. 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.

Transactional memory eliminates, by construction, many of the low-level difficulties that plague lock-based programming [6]. 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 little progress has been made on building transactional abstractions that compose well. We identify three particular problems.

Firstly, since a transaction may be re-run automatically, it is essential that it do nothing irrevocable. For example the transaction

```
atomic { if (n>k) then launch_missiles(); 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, blocking is not composable. Many systems do not support synchronisation at all without using condition variables, and those that do rely on a a programmer-supplied boolean guard on the `atomic` block [9]. 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 (but see Section 7.2 on Concurrent ML, which does). 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 [22] 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* [23].

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 – e.g. 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' }
```

A fuller introduction to concurrency, I/O, exceptions and cross-language interfacing (the "awkward squad" for pure, lazy, functional programming) is given in [21]. Several general on-line tutorials on Haskell are also available, for instance [3].

```
-- 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 ()
```

Figure 1: The `STM` interface

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 shall see, we are able to define a much more compositional form of transactional memory than has been possible hitherto. As we go, we will mention primitives from the `STM` library, whose interface is summarised in Figure 1, and whose semantics we will describe more thoroughly in Section 5.

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: in order 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 `atomic` function and all of the `STM`-typed operations are built over the transactional memory described in Section 6. This 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. 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 re-run with a fresh log.

Splitting the world into STM actions and I/O actions provides two valuable guarantees:

- Only STM actions and pure computation can be performed inside a memory transaction; in particular I/O actions cannot. 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). None of these mechanisms are composable.

The Haskell setting led us to a remarkably simple and composable mechanism for blocking: a single STM action `retry`. 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`. But if not, so there is insufficient resource in the variable, 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 `TVar`s read during the attempted transaction is written by another thread*. Furthermore, the transaction log (which is needed anyway) already records exactly which `TVar`s 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 `TVar`s involved, the producer will wake up the consumer. A whole class of lost-wake-up bugs is eliminated thereby.

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. *Any STM action can be robustly composed with other STM actions.*

3.4 Composing alternatives

We have discussed composing transactions in *sequence*, so that both are executed. STM Haskell also lets 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 it retries, then `s1` is abandoned with no effect, and `s2` is run. If `s2` retries as well, the entire call retries — but it waits on the variables read by *either* of the two nested transactions. Again, the programmer need know nothing about the enabling condition of `s1` and `s2`.

Using `orElse` provides an elegant way for library implementors 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
```

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 recognise 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. (As with `atomic`, no new language constructs are needed.) 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, the transaction is aborted with no effect. If the programmer wants to commit the effects up to the point at which the exception was thrown, he can easily catch the exception inside the transaction, and return normally — the transaction is only aborted if the exception propagates to the end of the `atomic` block.

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 between exceptions that cause the enclosing `atomic` block to abort from exceptions that allow it to commit before they are propagated. Shinnar *et al.* show how abort semantics are valuable when handling exceptions even in single-threaded applications [28].

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 block until 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"
   ; if 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`. On the other hand, since the transaction is aborted, no writes to `svar` are 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 store (or validation would fail, and the transaction would retry), 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 provide a full formal semantics in Section 5.

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. By aborting the transaction we provide a kill-safe mechanism for avoiding the kind of consistency problems that Flatt and Findler describe [5].

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 (for instance [10, 9]).

4.1 MVars

Prior to our STM work, Concurrent Haskell provided `MVar`s 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 `takeMVar` function leaves a full `MVar` empty, and blocks on an empty `MVar`. A `putMVar` on an empty `MVar` leaves it full, and blocks on a full `MVar`. So `MVar`s are, in effect, a one-place channel.

It is easy to implement `MVar`s on top of `TVar`s. 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 val
  = do { v <- readTVar mv
       ; case v of
            Nothing  -> writeTVar mv (Just val)
            Just val -> retry }
```

Notice how operations which 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

`MVar`s 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 the 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 two different multicast channels we can say:

$$
\begin{array}{rrcl}
 & x, y & \in & \textit{Variable} \\
 & r, t & \in & \textit{Name} \\
 & c & \in & \textit{Char} \\
\\
\text{Value} & V & ::= & r \mid c \mid \texttt{\textbackslash}x \texttt{->} M \\
 & & \mid & \texttt{return}\ M \mid M \texttt{>>=} N \\
 & & \mid & \texttt{putChar}\ c \mid \texttt{getChar} \\
 & & \mid & \texttt{throw}\ M \mid \texttt{catch}\ M\ N \\
 & & \mid & \texttt{retry} \mid M\ \texttt{`orElse`}\ N \\
 & & \mid & \texttt{forkIO}\ M \mid \texttt{atomic}\ M \\
 & & \mid & \texttt{newTVar} \\
 & & \mid & \texttt{readTVar}\ r \mid \texttt{writeTVar}\ r\ M \\
\\
\text{Term} & M, N & ::= & x \mid V \mid M\ N \mid \cdots
\end{array}
$$

Figure 2: The syntax of values and terms

```
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 which involves a mixture of `MVar`s and channels (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 them 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
```

This example is childishly simple in STM Haskell. In contrast, a function of type

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

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

4.4 Summary

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. More fundamentally, one can combine 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, again like high-level languages, 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.

$$
\begin{array}{rrcl}
\text{Thread soup} & P, Q & ::= & M_t \mid (P \mid Q) \\
\text{Heap} & \Theta & ::= & r \hookrightarrow M \\
\text{Allocations} & \Delta & ::= & r \hookrightarrow M \\
\\
\text{Evaluation} & \mathbb{E} & ::= & [\cdot] \mid \mathbb{E}\ \texttt{>>=}\ M \mid \texttt{catch}\ \mathbb{E}\ M \\
\text{contexts} & \mathbb{P} & ::= & \mathbb{E}_t \mid (\mathbb{P} \mid P) \mid (P \mid \mathbb{P}) \\
\text{Action} & a & ::= & \texttt{!}c \mid \texttt{?}c \mid \epsilon
\end{array}
$$

Figure 3: The program state and evaluation contexts

5. THE SEMANTICS OF STM HASKELL

So far our description of the functions in Figure 1 has been informal. It is hard to be sure that such descriptions cover all the combinations of these functions that might arise, so in this section we provide a formal, operational semantics for STM Haskell.

Figure 4 gives a small-step operational semantics for a small language whose syntax is given in Figure 2. The key idea is that there are *two* transition relations: the top-level *I/O transitions*, written " $\rightarrow$ "; and the *STM transitions*, written "$\Rightarrow$". The I/O transition relation takes a program state $P; \Theta$ to a new program state $Q; \Theta'$, while performing input/output described by an action a:

$$P; \Theta \xrightarrow{a} Q; \Theta'$$

Execution proceeds by repeatedly choosing a thread, and executing a single I/O transition; transitions from different threads may thereby be interleaved in a non-deterministic way. An `atomic` block, however, invokes zero or more steps of the STM transition relation, *but the result state change is regarded as a single I/O transition*; transitions in the STM relation therefore cannot interleave. The semantics has no notion of transaction logs or rollback – these are implementation matters. Instead the semantics expresses atomicity simply by requiring that an `atomic` block, if chosen for the next I/O transition, must reduce (using $\Rightarrow$) to a `return` or `throw`, and not to `retry`. The rest of this section fleshes out the details.

5.1 Syntax

Figure 2 gives the syntax of a fragment of STM Haskell. Terms and values are entirely conventional, except that we treat the application of monadic combinators, such as `return` and `catch`, as values. The `do`-notation we have been using so far is syntactic sugar for uses of `return` and `>>=`:

$$
\begin{array}{rcl}
\texttt{do}\ \{x\texttt{<-}e\texttt{;}\ Q\} & \equiv & e\ \texttt{>>=}\ (\texttt{\textbackslash}x\texttt{->}\ \texttt{do}\ \{Q\}) \\
\texttt{do}\ \{e\texttt{;}\ Q\} & \equiv & e\ \texttt{>>=}\ (\texttt{\textbackslash\_->}\ \texttt{do}\ \{Q\}) \\
\texttt{do}\ \{e\} & \equiv & e
\end{array}
$$

The monadic operations `return`, `>>=`, `throw`, and `catch` are overloaded, and can be used in both the `IO` and `STM` monad. Specific to the `IO` monad are:

```
getChar :: IO Char
putChar :: Char -> IO ()
forkIO  :: IO a -> IO ThreadId
```

I/O transitions are labelled with an optional action a, describing the input/output effect of the transition. The actions a (Figure 2) allow reading a character c from standard input `?`c, writing one to standard output `!`c, or the silent action ϵ, which is often omitted. A real system would have many more input/output actions.

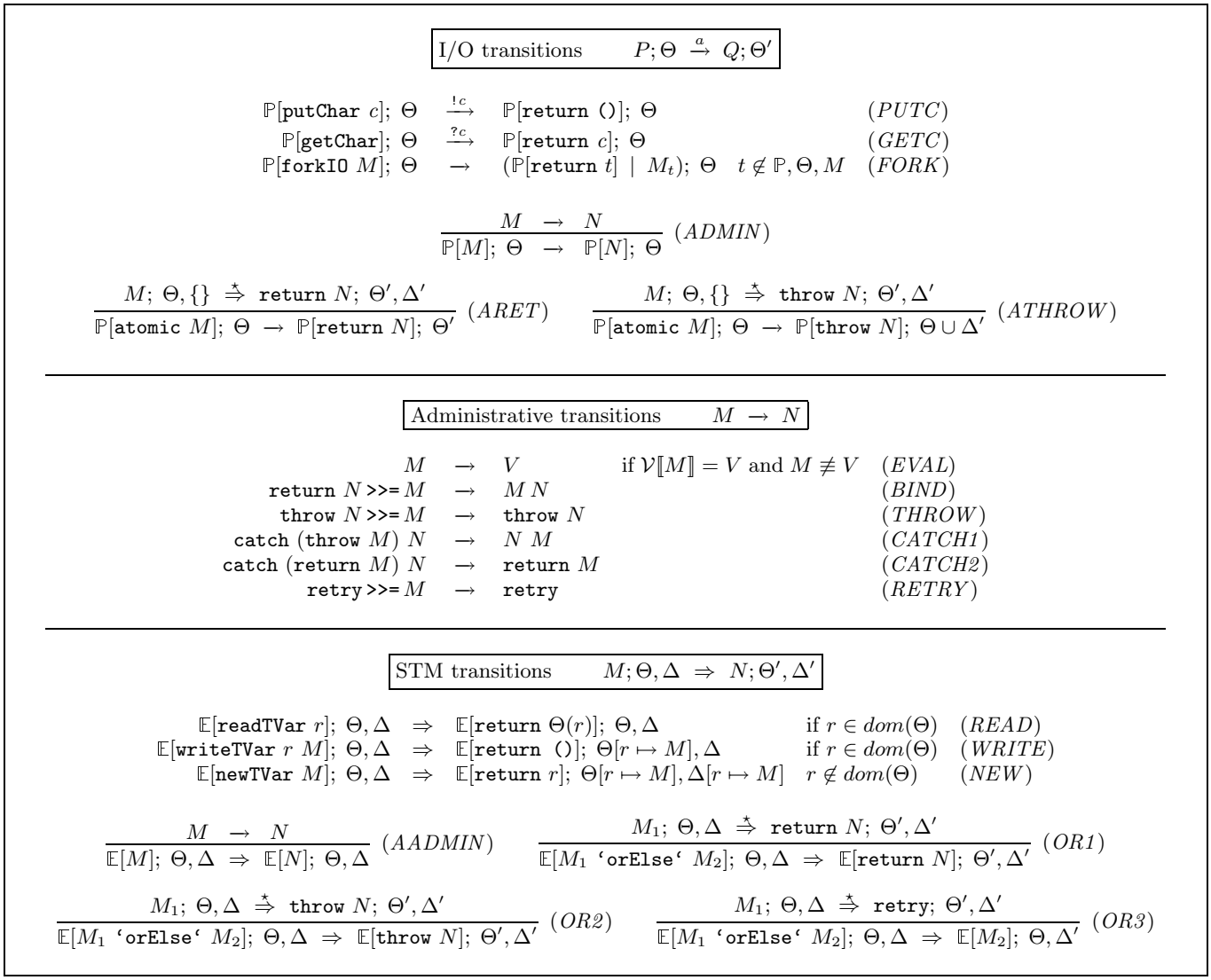


Figure 4: Operational semantics of STM Haskell

A *program state* $P;\Theta$ consists of a *thread soup* P and a *heap* Θ (Figure 3). A thread soup is just a multi-set of *threads*, each consisting of a single term M annotated with a thread ID t. A *heap*, Θ, is a finite mapping from *references* to terms.

To describe the possible transitions of a program state, we use an *evaluation context* to identify the active site for the transition. Figure 3 gives the syntax of evaluation contexts. A program evaluation context, $\mathbb{P}$, corresponds to the scheduler of a real implementation. It chooses an arbitrary thread from the soup, and then uses the term evaluation context $\mathbb{E}$ to find the active site in the term. The term evaluation context corresponds to the stack of a real machine, and looks into the left operand of `>>=`, `catch`, and `orElse`.

5.2 Operational semantics

Now we are ready to discuss the transition rules of Figure 4. First we treat the *I/O transitions*, in the top part of the figure, which can have arbitrary input/output effects. The first two rules deal with input and output. If the active term is a `putChar` or `getChar` the appropriate labelled transition takes place, and the operation is replaced by a `return` carrying the result. Rule (FORK) allows a new thread to be created, by adding a new term M to the thread soup, allocating a fresh name t as its `ThreadId`.

Rule (ADMIN) concerns *administrative transitions*, which are given in the second section of Figure 4. Rule (EVAL) allows a term M that is not a value to be evaluated by an auxiliary function, $\mathcal{V}[\![M]\!]$, which gives the value of M. This function is entirely standard, and we omit it here. Rule (BIND) implements sequential composition in the monad. The rules (THROW), (CATCH1) and (CATCH2) implement exceptions in the standard way. All of these rules are, as we shall see, used in both the `IO` monad and the `STM` monad, which is why we keep them in a separate group.

Everything so far is quite standard. The new part starts with rules (ARET) and (ATHROW). The former describes how an atomic transaction takes place: the term M makes zero or moretransitions of the STM relation, $\Rightarrow$, which takes the following form:

$$M;\ \Theta,\Delta \ \Rightarrow\ N;\ \Theta',\Delta'$$

Here, Θ is the heap as before, while Δ redundantly records the allocation effects (only) of the transition, for use during exception handling. Rule (ARET) specifies that the term M may make zero or more STM transitions until it reaches the form (`return` N), indicating successful completion. In that case, rule (ARET) takes one step, embodying the new heap Θ' as its resulting heap. In contrast, rule (ATHROW) specifies that if M evaluates to (`throw` N), then the new heap Θ' is discarded, and instead just the allocation effects Δ are added to the initial heap Θ.

Rules (ATHROW) and (ARET) are the only rules in the top panel of Figure 4 that affect the heap, so we can see immediately that the heap can be mutated only inside an `atomic` block. Furthermore, notice that *multiple STM transitions yield a single program transition*. Program transitions from different threads can be interleaved, but (ARET) provides no way for STM transitions to interleave. This is precisely what it means to execute "atomically". (A real implementation will not do this, but we are concerned with semantics here.)

The STM transitions themselves, in the last part of Figure 4, are largely standard. In particular, Rules (READ), (WRITE), and (NEW) describe how new mutable locations can be read, written, and created; the only point of interest is that (NEW) not only records the location's creation in the heap, but also in the allocation record Δ, for use by (ATHROW). Rule (AADMIN) lifts the administrative transitions into the STM world, just as t. The interesting part is the `orElse` combinator and `retry`, which we tackle next.

5.3 Blocking and nested transactions

The alert reader may be wondering why there is no rule (ARETRY) to go along with (ARET) and (ATHROW), to account for the fact that an STM computation may evaluate to `retry`, for instance:

```
atomic (do
  { v <- readTVar r
  ; if v==0 then retry else return ()
  ; ...})
```

What if `v` is zero? Then the body of the `atomic` block reduces to `retry`. *There is no rule for this case.* This means that the transition system cannot make progress by choosing a thread whose next operation is an `atomic` block, when the heap will cause it to `retry`. To make progress, another thread must be chosen.

Nested transactions are handled by rules (OR1-3). The first of these tries the left argument of an `orElse`. If it succeeds normally, then that is the result of the `orElse`, including any memory effects in Θ'. If it throws an exception, that too is the result of the `orElse`, and any memory effects are retained. But if M_1 retries, then rule (OR3) discards all its effects, and instead commits to M_2. Notice the strong similarity between (ARET) and (OR1), and between (ATHROW) and (OR2); this is the sense in which we say that `orElse` implements nested transactions.

An alternative design would have (OR2) behave like (OR3); that is, if M_1 throws an exception, we could discard its effects and try M_2 instead. But that would invalidate the beautiful identity which makes `retry` a unit for `orElse` and would also make `orElse` asymmetric in its treatment of exceptions (discarded from M_1 but retained for M_2). This was not a hard choice to make!

```
// Basic transaction execution
TLog *STMStart()
void *STMReadTVar(TLog *tlog, TVar *t)
void STMWriteTVar(TLog *tlog, TVar *t, void *v)

// Transaction commit operations
boolean STMIsValid(TLog *tlog)
void STMCommit(TLog *tlog)

// Blocking operations
void STMWait(TLog *tlog)
void STMUnWait(TLog *tlog)

// Nested-transaction operations
TLog *STMStartNested(TLog *outer)
void STMMergeNested(TLog *tlog)
```

Figure 5: The STM runtime interface

6. IMPLEMENTATION

Our implementation is split into two layers. The top layer implements the STM operations from Figure 1. This is built on top of the lower layer, which comprises a C library for performing memory transactions that is integrated in the Haskell runtime system. Figure 5 shows the API to our C library; we consider the four groups of operations in turn in Sections 6.1–6.4.

Concurrent Haskell is currently implemented only for a uni-processor. The runtime schedules lightweight Haskell threads within a single operating system thread. Haskell threads are only suspended at "safe points"; they cannot be pre-empted at arbitrary moments. This environment simplifies the implementation of our library because, by construction, C runtime functions run without interruption.

We are confident that a multi-processor implementation is practical: our previous work has developed several techniques for building multi-processor STMs in which a multi-word atomic update is no worse than half the speed of a uniprocessor design [10, 9]. These have been tested in practice on 1..96-CPU shared memory machines, giving scalable performance when threads are attempting non-conflicting transactions (for instance, concurrent inserts on different parts of a red-black tree could commit in parallel). Even in the intensive workload we describe in Section 6.6, the commit operation is less than 10% of total execution time and so the overall consequences of using a parallel version would be low.

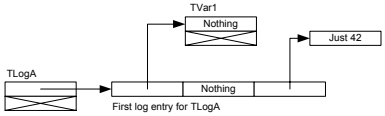
6.1 Transaction logs and TVar accesses

While executing a memory transaction, a thread-local *transaction log* is built up recording the reads and tentative writes that the transaction has performed. This transaction log is held in a heap-allocated object called a `TLog` that is pointed to by the Thread Control Block of the thread engaged in the transaction.

The log contains an entry for each of the `TVar`s that the memory transaction has accessed. Each entry contains a reference to the `TVar` involved, the *old value* held in the `TVar` when it was first accessed in the transaction, and the *new value* to be stored in the `TVar` if the transaction commits. These two values are identical in the case a `TVar` that has been read but not written by the transaction.

Within a transaction, all `TVar` accesses are performed by `STMReadTVar` and `STMWriteTVar` (Figure 5). These accesses

(a) A `TVar` representing a single-cell buffer. The value `Nothing` indicates that the buffer is empty. The `TVar`'s queue of waiting threads is empty.



(b) A transaction log (`TLogA`) containing a tentative update to `TVar1`. The transaction proposes replacing `Nothing` with `Just 42` to indicate that the buffer holds `42`.

Figure 6: Transaction logs

remain buffered within the thread's log, and hence invisible to other threads, until the transaction commits (Section 6.2): writes are made to the log, and reads first consult the log so that they see preceding writes from the same transaction. Hence, a transaction can be aborted with no effect simply by discarding its log.

Figure 6 shows the structure of `TLog`s and `TVar`s. It depicts a transaction executing the code from Section 4.1 that builds an `MVar` buffer using a `TVar`. In (a) the `TVar` refers to the value `Nothing` indicating that the buffer is empty. In (b) the thread reads from the `TVar`, sees it to hold `Nothing` and creates a new log entry that tentatively places the value 42 in the buffer. The fields depicted with a big cross, indicating null, are discussed in subsequent sections.

Our transaction logs are ordinary heap-allocated structures. This means that we can rely on the garbage collector to avoid A-B-A problems.

6.2 Validation and commit

The `atomic` function operates by pushing an `AtomicFrame` entry onto the Haskell stack, and invoking `STMStart` to allocate a fresh transaction log (Figure 5). When execution returns to this frame the log is *validated*, using `STMIsValid`, to check that it reflects a consistent view of memory. For each log entry, validation checks that the old value is pointer-equal to the current contents of the `TVar`. If validation succeeds, `STMCommit` is called to apply the changes to the heap. Otherwise, the `TLog` is discarded, a fresh transaction is started and the atomic block re-executed. This entire validate-and-then-commit sequence is carried out atomically with respect to all other threads – see the remarks at the start of Section 6.

If an exception propagates to the `AtomicFrame` (ATHROW in Figure 4) then, rather than just abandoning the transaction, we must still call `STMIsValid`. This ensures that the transaction saw a consistent view of memory, and re-executes it if not. Why? Because it is entirely possible that the exception was thrown solely *because* the transaction saw an inconsistent view of memory, and the programmer must never know that this has happened.

In fact, it is also possible that an inconsistent view of memory might lead to non-termination. For example, consider:

```
f :: Integer -> Bool
f x = if x==0 then True else f (n-1)
```

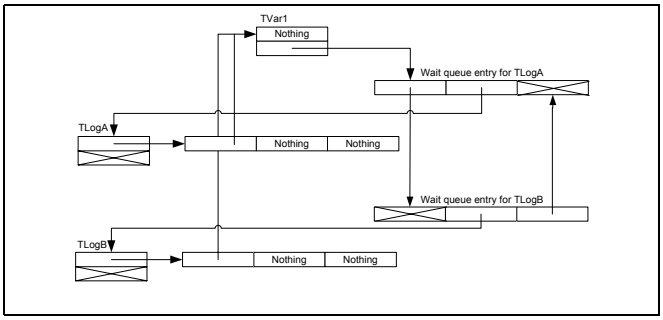


Figure 7: Logs for two threads blocked on `TVar1`

```
foo = atomic (do
         { x <- readTVar v
         ; y <- readTVar v
         ; if f (x-y) then ... else ... })
```

If `foo` saw memory at a moment at which `x-y` was less than zero, the call to `f` would loop infinitely. Nontermination is an effect that the type system does not track!

Our solution to this is simple: whenever the scheduler is about to switch to a thread that is engaged in a transaction, the scheduler first calls `STMIsValid` to check that the transaction is not already doomed. If it is, the stack is unrolled back to the `AtomicFrame` and the transaction is re-started. In this way, doomed transactions can be killed off before they have consumed too much time. It does not make sense to validate more frequently on a uniprocessor (indeed, less frequently might perform better) but, as in previous work, we might use an alternative scheme on a multiprocessor.

6.3 Blocking transactions: retry

Leaving aside the possibility of `orElse` for the moment, calling `retry` causes the stack to be unwound searching for the enclosing `AtomicFrame` — the types guarantee that exactly one such frame exists. Then `STMIsValid` is called, as usual, to check that the transaction log has seen a consistent view of the heap, and if not the transaction is re-run. In the consistent case, `STMWait` is called. It allocates new *wait-queue entries*, held in doubly-linked lists attached to the `TVar`s that the transaction has read, using the previously-null field in each `TVar`. Once this is done, the calling thread is responsible for blocking itself and re-entering the scheduler.

The wait queue entries are noticed by an `STMCommit` which updates the `TVar`s: the updater unblocks any waiters it encounters. Once a waiting thread is rescheduled, it is responsible for calling `STMIsValid` to assess whether it should retry execution of its atomic block. If the transaction is no longer valid then `STMUnWait` unlinks its wait queue entries and the caller retries its transaction with a fresh log. If the transaction is still valid then it leaves its wait queue entries in place, so that it can be woken by further updates, and blocks once more – this can happen only if, by the time the thread is scheduled, the `TVar`s again contain pointer-equal values to those originally read by that thread. Figure 7 illustrates this, depicting two threads both waiting for `TVar1` to be updated.

6.4 Nested transactions: orElse

The final piece of the implementation is `orElse`, which places two additional requirements on the STM. Firstly, proper nested transactions are needed, to isolate the execution of

the two alternatives: if the first alternative retries, any updates it has proposed must be invisible when trying the second alternative. Nesting is handled by `STMStartNested` which creates a fresh log for a nested transaction. While executing a nested transaction, writes are recorded (only) in the nested transaction's log, while reads must consult *both* the nested log *and* the logs of its enclosing transactions.

If either alternative completes without retrying then the nested transaction is validated by calling `STMIsValid` to check that it has seen a consistent view of the heap. Validating a nested transaction requires us to also validate its enclosing transactions: if any of them has become invalid by a concurrent update then we re-execute the whole `atomic` function with a fresh log. If the nested transaction is valid then `STMMergeNested` is called. This examines each entry in the nested log: if the parent already contains an entry for the `TVar` involved then the *new value* (only) is copied from the nested log, otherwise the entire entry is copied.

If both alternatives call `retry` then we propagate the retry in such a way that the thread will wait on the *union* on the sets of `TVar`s that they have accessed. To do this, we first validate the logs for both nested transactions. If either is invalid then we re-execute the outer transaction with a fresh log. Otherwise, if both are valid, we call `STMMergeNested` on the two logs, in either order. Figure 8 illustrates this for the case of `orElse` being used to combine operations on two `TVar`s. In (a) the two nested transactions hold the accesses made within the two branches of `orElse`. In (b) these nested transactions have been merged to their parent so that, if the retry propagates to the `AtomicFrame`, the thread will block on the union of the sets of `TVar`s involved.

Note the "in either order". There is a subtle question about what happens when the two alternatives supplied to `orElse` try to perform conflicting updates before retrying:

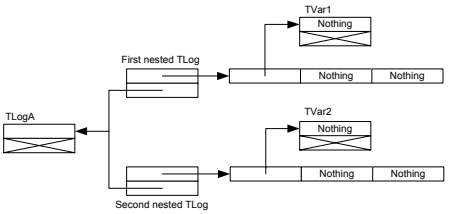
```
do { writeTVar v 10 ; possiblyRetry }
`orElse`
do { writeTVar v 20 ; possiblyRetry }
```

If both alternatives retry then their logs will hold inconsistent updates to `v`, so the final merged log will contain either 10 or 20 as the *new value* for `v`, depending on which log is the last to be merged. However, when retrying, the *new value* in the merged log does not matter: the log will be subject only to further merges, or eventually to `STMWait`.

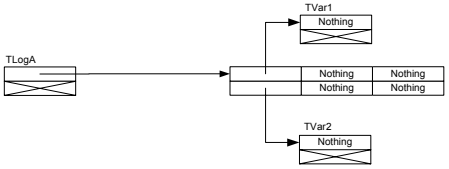
6.5 Progress

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 to `TVar` y, while a second reads and writes to those `TVar`s in the opposite order. Each transaction would observe the original value of those `TVar`s, the first to validate will commit, and the second will abort and restart. Similarly, synchronisation conflicts over `TVar`s cannot cause cyclic restart, where two or more transactions repeatedly abort one another.

Starvation is possible. For example, a transaction that runs for a very long time may repeatedly conflict with shorter transactions. 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.



(a) Initially the two alternatives combined by `‘orElse‘` are run in separate nested transactions.



(b) If both branches `retry` then the two logs are merged into the enclosing transaction and the `retry` propagates.

Figure 8: Two steps in the implementation of `‘orElse‘`

6.6 Performance

Evaluation of the STM implementation described here is at an early stage, so there are no detailed performance results to report as yet.

However, initial measurements are encouraging. We wrote a simple implementation of unbounded channels using STM Haskell, which mirrors the MVar-based channels in Concurrent Haskell [22]. We benchmarked the two implementations by measuring the time taken to communicate a large number of values over a channel between two threads. They performed almost identically: runtimes were the same (to within 10%), and the STM version allocated 50% less heap space during the run.

Why should this be the case, given that the STM version appears to be doing more bookkeeping under the hood? The raw MVar operations would outperform the equivalent TVar operations if we benchmarked them independently, but in practice programs don't perform raw MVar operations. Instead, the MVar operation is normally wrapped in an exception handler that restores invariants in the event of an exception. Further protection from asynchronous exceptions is usually required, to prevent an asynchronous exception from arriving before the handler has been installed [17]. This exception-robustness is implemented in the `MVar`-based channel library that we used, but it adds significant overhead to MVars.

In contrast, our STM code benefits from asynchronous exception safety "for free", because each channel operation is atomic. In short, the STM-based channels are not only clearer, but the operations are composable, and it runs just as fast as the MVar version.

7. RELATED WORK

We build on two main categories of related work. The first, discussed in Section 7.1, is work on transactional models of concurrency and the design and implementation of STMs. The second, in Sections 7.2–7.3 are the designs that have been attempted to provide forms of composability in concurrent programming languages.

7.1 Transactions

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.

Nested transactions were first proposed by Moss [19], who extended nesting to two-phase locking protocols. The Argus language [16] for fault-tolerant distributed applications provided explicit language support for nested transactions.

Distributed transactions typically provide both *synchronisation*, ensuring that concurrently-executing transactions appear to execute serially, and *persistence*, ensuring that state changes are backed up on fault-tolerant, non-volatile storage. Recently, several projects have provided persistence without synchronisation for transactions running at a single machine [15, 26, 13].

By contrast, software transactional memory provides synchronisation without persistence. Because the state manipulated by memory transactions is intentionally volatile 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.

Transactional memory was originally proposed as a hardware architecture [11, 29] to support non-blocking synchronisation, and architectural support for this model remains the subject of ongoing research [18, 20, 24, 8]. A number of proposals have emerged for supporting transactional memory in software [12, 27, 4, 10, 9].

Work on software transactional memory has focused on libraries, not on integrating transactional mechanisms into a programming language. Two exceptions are Welc *et al.* [31] who show how STM-like techniques can increase the concurrency available in systems based on Java's `synchronized` blocks, and Harris and Fraser [9] who discuss how Java might be adapted to support non-blocking atomic sections. In recent work Welc *et al.* showed how I/O could be performed by backing off from an optimistic execution scheme to a pessimistic one – however, their approach relied on starting with a correctly-synchronized lock-based program [30].

Prior work has not placed much emphasis on conditional blocking or compositionality. Herlihy *et al.* [10] support syntactically nested transactions by "flattening" nested transactions to a single transaction, but provide no explicit mechanism for conditional blocking. Harris and Fraser [9] support conditional blocking using a guarded-command syntax, but lacking `retry`, such transactions could not be easily composed. Lastly, no prior work on memory transactions supports the equivalent of the `orElse` construct, which is essential for composition.

7.2 Concurrent ML

Concurrent ML [25] is an inspiring language directed squarely at the goal of composable concurrency. The principal abstraction is that of a *first-class event*, which allows far richer composition than do conventional locks. One can draw an analogy between a CML event and an STM action in our language. Events can be composed as alternatives using `choose`, which is similar to our `orElse`, and "run" using `sync`, which has the same flavour as our `atomic`; in Haskell syntax their types are:

```
sync   :: Event a -> a
choose :: [Event a] -> Event a
```

However, nothing corresponds to our notion of sequential composition of actions. Indeed, given an `Event a` and an `Event b`, one cannot construct a compound event of type `Event (a,b)` that fires only when both argument events fire. This is no accident — CML events are carefully structured to have a single "commit point" — but it limits the way in which events can be composed.

This same limitation does support one form of abstraction that we cannot. A *swap channel* offers the operation

```
swap :: SwapChan a -> a -> Event a
```

The idea is that two threads rendezvous at a `SwapChan`, and exchange data. But no matter how many threads are simultaneously calling `swap` on the same channel, if thread A gives data to thread B, then B's data must go to A. We cannot support a composable `swap` inside an STM transaction because that would require mutual linkage of an arbitrary number of threads whereas STM actions represent isolated updates made by individual threads. Suppose thread A does a `swap` with thread B; and then both go on to `swap` with third parties (A1 and B1, say). Then if A1 is not ready, A's transaction must retry; and hence so must B's, and so must B1's, and so on. In contrast, it is easy to define swap-channels with the operation

```
swap :: SwapChan a -> a -> IO a
```

but this operation, having an `IO` type, does not compose (by design). It is perhaps interesting to note for future work that this kind of synchronization, which is hard to build with STM, is extremely easy to build with a *chord* in Benton *et al.*'s Polyphonic C# [1].

7.3 Scheme48 proposals

Scheme 48 *proposals* are an optimistic-concurrency mechanism that supports a subset of our notion of memory transactions [14]. Each thread maintains a log of the data accesses performed using operations like `provisional-car`. The call `call-ensuring-atomicity t` is just like our `atomic t`; it re-runs automatically if there are concurrent updates.

Of course, Scheme is untyped, so the proposal mechanism cannot offer any guarantees about effects; for example, there is no way to ensure that the programmer only uses `provisional-car` etc inside a transaction, nor that transactions refrain from doing input/output. There is no mechanism for conditionally entering a proposal (and blocking if the condition does not hold), let alone for our modular `retry`. The programmer must resort to locks and condition variables for that. Nor is there anything like `orElse`.

8. CONCLUSION

We have shown that STM provides a substrate for concurrent programming that offers far richer composition than has been available to date, and that it can be implemented in a practical language.

We have used Haskell as a particularly-suitable laboratory, but an obvious question is this: to what extent can our results be carried back into the mainstream world of imperative programming? We believe that the idea of using constructs like `retry` and `orElse` can indeed be applied to other languages. For instance, in C#, one could indicate `retry` by raising a specified kind of exception and then express `orElse` as a particular kind of exception handler.

An interesting distinction to notice about `atomic` blocks in C# or Java, when compared with Haskell, is that it would be necessary to support dynamic nesting. The reason is that, in Haskell, the code within an `atomic` block has an `STM` type and so the *only* way it can be run is by atomic execution: library operations do not need to ensure atomicity internally because it will be provided by their callers. In contrast, in a traditional imperative language, atomicity would be the responsibility of the callee rather than the caller and so it may be provided defensively at multiple levels in a call chain.

In an imperative setting it is less clear how to statically prevent operations with irreversible side effects being performed within transactions: there is not ordinarily any way of indicating possible effects other than (in some languages) the sets of exceptions that a method may raise. Whether or not one believes in transactions, it does seem likely that some combination of effect systems and/or ownership types [2] will play an increasingly important role in concurrent programming languages, and these may contribute to the guarantees desirable for memory transactions.

Our implementation forms part of GHC 6.4, which is publicly available at `http://haskell.org/ghc`. Our current implementation is for uni-processor, but we plan to work on a true multi-processor implementation in 2005.

Acknowledgments

We would like to thank Byron Cook, Austin Donnelly, Jim Gray, Matthew Flatt, 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 this paper. Special thanks to Andres Löh for help with typesetting the figures.

9. REFERENCES

[1] Benton, N., Cardelli, L., and Fournet, C. Modern concurrency abstractions for C♯. In *Proceedings of European Conference on Object-Oriented Programming* (June 2002), vol. 2374 of *Lecture Notes in Computer Science*, pp. 415–425.

[2] Clarke, D. G., Potter, J. M., and Noble, J. Ownership types for flexible alias protection. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (1998), pp. 48–64.

[3] Daume III, H. Yet Another Haskell Tutorial. 2004. Available at `http://www.isi.edu/~hdaume/htut/` or via `http://www.haskell.org/`.

[4] Ennals, R. Feris: a functional environment for retargetable interactive systems. Batchelors thesis, University of Cambridge Computer Laboratory, 2000.

[5] Flatt, M., and Findler, R. B. Kill-safe synchronization abstractions. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation* (2004), pp. 47–58.

[6] Fraser, K. *Practical lock-freedom*. PhD thesis, University of Cambridge Computer Laboratory, Feb. 2004. Also published as UCAM-CL-TR-579.

[7] Gray, J., and Reuter, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1992.

[8] Hammond, L., Carlstrom, B. D., Wong, V., Hertzberg, B., Chen, M., Kozyrakis, C., and Olukotun, K. Programming with transactional coherence and consistency (TCC). In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems* (2004), pp. 1–13.

[9] Harris, T., and Fraser, K. Language support for lightweight transactions. In *Proc ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications (OOPSLA)* (2003), pp. 388–402.

[10] Herlihy, M., Luchangco, V., Moir, M., and Scherer, III, W. N. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing* (2003), pp. 92–101.

[11] Herlihy, M., and Moss, J. E. B. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture* (1993), pp. 289–300.

[12] Israeli, A., and Rappoport, L. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing* (1994), pp. 151–160.

[13] Jordan, M., and Atkinson, M. Orthogonal persistence for Java — a mid-term report. In *Proceedings of the 3rd International Workshop on Persistence and Java* (Sept. 1998), pp. 335–352.

[14] Kelsey, R., Rees, J., and Sperber, M. The incomplete scheme 48 reference manual for release 1.1. July 2004.

[15] Lamb, C., Landis, G., Orenstein, J., and Weinreb, D. The ObjectStore database system. *Commun. ACM 34*, 10 (1991), 50–63.

[16] Liskov, B. Distributed programming in Argus. *Commun. ACM 31*, 3 (1988), 300–312.

[17] Marlow, S., Peyton Jones, S., Moran, A., and Reppy, J. Asynchronous exceptions in Haskell. In *ACM Conference on Programming Languages Design and Implementation (PLDI'01)*, ACM, pp. 274–285.

[18] Martnez, J. F., and Torrellas, J. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *Proceedings of the 10th international conference on architectural support for programming languages and operating systems (ASPLOS-X)* (2002), pp. 18–29.

[19] Moss, E. B. Nested transactions: An approach to reliable distributed computing. Tech. rep., Massachusetts Institute of Technology, 1981.

[20] Oplinger, J., and Lam, M. S. Enhancing software reliability with speculative threads. In *Proceedings of the 10th international conference on architectural support for programming languages and operating systems (ASPLOS-X)* (2002), pp. 184–196.

[21] Peyton Jones, S. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction, Marktoberdorf Summer School 2000*, C. Hoare, M. Broy, and R. Steinbrueggen, Eds., NATO ASI Series, IOS Press, pp. 47–96.

[22] Peyton Jones, S., Gordon, A., and Finne, S. Concurrent Haskell. In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pp. 295–308.

[23] Peyton Jones, S., and Wadler, P. Imperative functional programming. In *20th ACM Symposium on Principles of Programming Languages (POPL'93)*, pp. 71–84.

[24] Rajwar, R., and Goodman, J. R. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th international conference on architectural support for programming languages and operating systems (ASPLOS-X)* (2002), pp. 5–17.

[25] Reppy, J. *Concurrent programming in ML*. Cambridge University Press, 1999.

[26] Satyanarayanan, M., Mashburn, H. H., Kumar, P., Steere, D. C., and Kistler, J. J. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems 12*, 1 (1994), 33–57.

[27] Shavit, N., and Touitou, D. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing* (1995), pp. 204–213.

[28] Shinnar, A., Tarditi, D., Plesko, M., and Steensgaard, B. Integrating support for undo with exception handling. Tech. Rep. MSR-TR-2004-140, Microsoft Research, Dec. 2004.

[29] Stone, J. M., Stone, H. S., Heidelberger, P., and Turek, J. Multiple reservations and the Oklahoma update. *IEEE Parallel and Distributed Technology 1*, 4 (Nov. 1993), 58–71.

[30] Welc, A., Hosking, A. L., and Jagannathan, S. Preemption-based avoidance of priority inversion for java. In *Proceedings of the 2004 International Conference on Parallel Processing (ICPP)* (Aug. 2004), pp. 529–538.

[31] Welc, A., Jagannathan, S., and Hosking, A. Transactional monitors for concurrent objects. In *Proceedings of the European Conference on Object-Oriented Programming* (June 2004).