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 (STM) which allows groups of heap operations to be performed atomically [25, 8, 7, 4]. Using transactional memory (implemented using optimistic synchronisation) instead of locks brings several well-known advantages: freedom from deadlock and priority inversion, robust and automatic roll-back after exceptions or timeouts, and freedom from the usual 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, by focusing on multi-word atomic updates, 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 3.1). This is much more than a routine "port" of existing work 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.

- 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 support necessary for STM is localised, rather than pervasive, and we describe the details in Section 6.

We are very excited about this work. 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 compositional-ity: a programmer can control atomicity and blocking behaviour in a modular way that respects abstraction barriers. In contrast, with much current technology, abstraction and concurrency are in direct conflict (Section 2).

1. Introduction

Concurrent programming is notoriously difficult. 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, or being able to impose additional locking at all of the other places in which a shared object is accessed.

To address some of these difficulties, researchers have proposed the use of software transactional memory (STM) which allows atomic groups of heap operations to be performed atomically [25, 8, 7, 4].

Using transactional memory (implemented using optimistic synchronisation) instead of locks brings several well-known advantages: freedom from deadlock and priority inversion, robust and automatic roll-back after exceptions or timeouts, and freedom from the usual 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, by focusing on multi-word atomic updates, 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 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 is essential for allowing 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.

- We re-express the ideas of transactional memory in the setting of Concurrent Haskell (Section 3.1). This is much more than a routine “port” of existing work 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.

- 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 support necessary for STM is localised, rather than pervasive, and we describe the details in Section 6.

We are very excited about this work. 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 compositional-ity: a programmer can control atomicity and blocking behaviour in a modular way that respects abstraction barriers. In contrast, with much current technology, abstraction and concurrency are in direct conflict (Section 2).

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 notoriously 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 that supports 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. Yet more complexity is required if the client wants to first awaiting 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 readily be composed into larger correct operations.

The same phenomenon shows up for parallel composition. 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 descriptors that must be waited for, do 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:

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

and to wait for either p1 or p2 we can say

```haskell
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 transactional memory is not new. Transactions have been a fundamental mechanism in database design for many years, but the idea of applying the same idea to memory is more recent [9, 8, 7, 27, 4].

The key idea is that a block of code, including nested calls, can be enclosed by an atomic block, with the guarantee that the enclosed block 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 performs. 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 method’s execution, then the block is re-executed from scratch.

Transaction memory eliminates, by construction, many of the low-level difficulties that plague lock-based programming [4]. 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

```haskell
atomic { if (n>m) 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 m, 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 transactional-memory systems do not support synchronisation at all, and those that do rely on a programmer-supplied boolean guard on the atomic block [7]. For example, a method to get an item from a buffer might look like this:

```haskell
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. Nor can we use atomic { get(); get() }, because the pre-conditions for atomic in (say) the second get() might not hold; and we can’t block there awaiting items to be added, because then we would not get consecutive items. The only way out is to have an enclosing atomic block whose pre-condition checks that 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 separate or different 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 [20] is an extension to Haskell 98, a pure, lazy, functional programming language. It provides explicitly-forked threads, and abstractions for communicating between them. These constructs naturally involve 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 using 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 input/output before yielding a value of type a. For example, the functions putStrLn and getChar have types:

```haskell
putStrLn :: String -> IO ()
getChar :: IO Char
```

That is, putStrLn takes a String and delivers an I/O action that, when performed, prints the string 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:

```haskell
main :: IO ()
main = putStrLn '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:

```haskell
main = do c <- getChar; putStrLn c; putStrLn 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:
are explicit, and relatively rare: most computation takes place in the input/output effects. Second, reads from and writes to mutable cells so that transactions can perform memory effects but not irreversible effects from effect-free ones. As we shall see, it is easy to refine it to a mutable storage location and returns the value that it contains.

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

```
newIOLRef :: a -> IO (IOLRef a)
readIOLRef :: IOLRef a -> IO a
writeIOLRef :: IOLRef a -> a -> IO ()
```

evalRef takes a value of type a and creates a mutable storage location holding that value. readIOLRef takes a reference to a mutable storage location and returns the value that it contains. writeIOLRef provides the corresponding update operation. Since mutable 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 function `sin` with type `Float -> Float` cannot read or write an IOLRef, and that fact is guaranteed by its type.

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

```
-- 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 ()
   orElse :: STM a -> STM a -> STM a

   -- Programing STM
   check :: Bool -> STM a
   check True = return ()
   check False = retry

   -- 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

```
newIOLRef :: a -> IO (IOLRef a)
readIOLRef :: IOLRef a -> IO a
writeIOLRef :: IOLRef a -> a -> IO ()
```

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 of type \(\text{IO} t\) 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 irrecoverable 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 simple resource manager, which holds an integer-valued resource. The call `getR r n` should acquire units of resource \(r\), blocking if \(r\) holds insufficient resource; the call `putR r n` should return \(n\) units of resource \(r\).

Suppose we wanted to program a integer-valued resource, with operations to put and get units of resource. 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); ... }
```

Operationally, `atomic` takes the tentative updates and actually applies them to the `TVars` involved, thereby exposing these effects to other transactions. The `atomic` function and all of the STM-typed operations are built over the software transactional memory described in Section 6. This deals with maintaining a per-thread transactional log to record the tentative accesses made to `TVars`. 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. 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, because it statically prevents the programmer from calling `launchMissiles` inside a transaction, because `launchMissiles` 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 write...
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 whose type is just STM a. Here is the code for getR:

```haskell
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 TVars read during the attempted transaction is written by another thread. Furthermore, the transaction log (which is needed anyway) already records exactly which TVars were read. The implementation blocks the thread until at least one of these is updated.

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

Notice that there is no need for the getR 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 eliminated thereby.

From an efficiency point of view it makes sense to put calls to retry as early as possible in the transaction, and to refrain from reading locations that are not required for the test until after the test succeeds. Nevertheless, the programming interface is delightfully simple, and easy to reason about.

3.3 Composing memory transactions sequentially

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 could say

```haskell
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 absolutely central to making memory transactions composable even when they may block. The transaction above will block if either r1 or r2 has insufficient resource. There is no need for the programmer to know how getR is implemented, or what condition guarantees its success. Nor is there any concern that the thread may block awaiting r2 while holding r1, with the attendant risk of deadlock.

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 allows us to compose transactions as alternatives, so that one or the other, but not both, is executed. For example, to get either 3 units from r1 or 7 units from r2, we can say

```haskell
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. In the above example, the entire transaction retries if neither r1 nor r2 has sufficient resource — but it waits on the variables read by either of the two nested transactions. So when either resource is written to the transaction will be re-run. Again, the programmer need know nothing about the enabling condition of the arguments to orElse.

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:

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

Notice that this function 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.

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

\[
M _1 \text{orElse} (M _2 \text{orElse} M _3) = (M _1 \text{orElse} M _2) \text{orElse} M _3
\]

\[
M \text{orElse} M = M
\]

\[
M \text{orElse} retry = M
\]

3.5 Synchronous exceptions

The STM monad supports exceptions just like the IO monad, and in much the same way as (say) C#. The main functions – as with atomic, no new language constructs are required – are catch and throw, whose types are given in Figure 1. The question is: how should transactions and exceptions interact. For example, what should this transaction do?

```haskell
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... }
```

Haskell aficionados will recognise that STM is thereby an instance of MonadPlus.
The programmer throws an `AssertionFailed` exception if \( n > \text{lim} \), in which case the `.write data.` part will clearly not take place. But what about the write to \( v \cdot n \), which took place before the exception was thrown?

Concurrent Haskell encourages programmers to use exceptions for signaling error conditions, rather than for 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 \( v \cdot n \) 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 [?].

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

```haskell
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 (AssertionFailed s), where s is the string 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 deal with this case 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 committed; after all, they cannot affect any other thread. 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.

### 3.6 Asynchronous exceptions

An asynchronous exception is a signal from outside the thread, usually thrown from some other thread using `throwTo`:

```haskell
throwTo :: ThreadId -> Exception -> IO ()
```

A typical example is when a master thread wishes to shut down, and kill its helper threads, or when a thread wishes to perform an operation subject to a timeout and arranges for a second thread to send it an asynchronous exception when the time expires. Other possibilities include stack overflow, or some other resource exhaustion. In any case, the thread has no way to know when an asynchronous exception might strike.

If a thread is in the midst of a STM transaction when an asynchronous exception arrives, the transaction is aborted, with no externally-visible effects. A thread cannot throw an asynchronous exception (to another thread, to itself) from the STM monad, because doing so is an irreversible effect; and the type of `throwTo` enforces this constraint. By aborting the transaction we provide a kill-safe mechanism for avoiding the kind of consistency problems that Flatt and Findler describe [3].

## 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 [8, 7]).

### 4.1 MVars

Our previous design, Concurrent Haskell, provides MVars as its 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 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”).

```haskell
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:

```haskell
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:

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

Notice how operations which returns a boolean success / failure result can be built directly from these blocking designs. For instance:

```haskell
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. We can extend this to form 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:

```haskell
data MChan 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:

```haskell
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:

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

With these definitions, the code writes itself:

```haskell
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 another when the buffer is empty); 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:

```haskell
atomic (readPort pl `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 TVar at the extremity of each channel.

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

```haskell
atomic (readPort pl `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 merged. 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:

```haskell
merge :: [STM a] -> STM a merge = foldr1 orelse
```

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

```haskell
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.

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 \( a \):

```haskell
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 `>>=`:

\[
\text{do } \{ x \leftarrow e; Q \} \equiv e \gg= x \rightarrow \text{do } \{ Q \}, \\
\text{do } \{ e; Q \} \equiv e \gg= \_\rightarrow \text{do } \{ Q \}, \\
\text{do } \{ e \} \equiv e
\]

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 \rightarrow IO ()`
- `forkIO :: IO a \rightarrow IO ThreadId`

I/O transitions are labeled 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 \( i \rightarrow c \), writing one to standard output \( o \), or the silent action `e`, which is often omitted. A real system would have many more input/output actions.

A program state \( P, \Theta \) consists of a thread soup \( P \) and a heap \( \Theta \) (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, \( \Theta \), 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, \( E \), corresponds to the scheduler of a real implementation. It chooses an arbitrary thread from the soup, and then uses the term evaluation context \( E \) to find the active site in the term. The term

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 labeled 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, \( V[M] \), 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 more\(^2\) transitions of the STM relation, \( \Rightarrow \), which takes the following form:

\[
M; \Theta, \Delta \Rightarrow N; \Theta', \Delta'
\]

Here, \( \Theta \) is the heap as before, but \( \Delta \) 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 \( \text{return } N \), indicating successful completion. In that case, rule (ARET) takes one step, embodying the new heap \( \Theta' \) as its resulting heap. In contrast, rule (Athrow) specifies that if \( M \) evaluates to \( \text{throw } N \), then the new heap \( \Theta' \) is discarded, and instead just the allocation effects \( \Delta' \) are added to the initial heap \( \Theta \).

Rules (Athrow) and (ARET) are the only rules in the top part of Figure 4 that affect the heap, so we can see immediately that the heap can only be mutated 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) records its result in the allocations record \( \Delta \), which is used 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 example, consider

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

\(^2\) The repetition is indicated by the star.
I/O transitions \[ P;\Theta \xrightarrow{\tau} Q;\Theta' \]

\[ P[\text{return }c];\Theta \xrightarrow{\tau} P[\text{return }();\Theta] \quad (\text{PUTC}) \]
\[ P[\text{getChar}];\Theta \xrightarrow{\tau} P[\text{return }c];\Theta \quad (\text{GETC}) \]
\[ P[\text{fork10 }M];\Theta \xrightarrow{\tau} (P[\text{return }t \mid M];\Theta \quad t \notin P,\Theta \quad (\text{FORK}) \]

\[ M \xrightarrow{\tau} N \quad (\text{ADMIN}) \]
\[ M;\Theta,\Delta \xrightarrow{\tau} \text{return }N;\Theta',\Delta \quad (\text{ARET}) \]
\[ M;\Theta,\Delta \xrightarrow{\tau} \text{throw }N;\Theta',\Delta \quad (\text{ATHROW}) \]

Administrative transitions \[ M \xrightarrow{\tau} N \]

\[ \text{return }N \Rightarrow M \xrightarrow{\tau} MN \quad (\text{EVAL}) \]
\[ \text{throw }N \Rightarrow M \xrightarrow{\tau} \text{throw }N \quad (\text{BIND}) \]
\[ \text{catch }\text{(throw } M)\rightarrow N \xrightarrow{\tau} NM \quad (\text{THROW}) \]
\[ \text{catch }\text{(return } M)\rightarrow N \xrightarrow{\tau} \text{return }M \quad (\text{CATCH1}) \]

STM transitions \[ M;\Theta,\Delta \xrightarrow{\tau} N;\Theta',\Delta' \]

\[ E[\text{readTV }r];\Theta,\Delta \Rightarrow E[\text{return }\Theta(r)];\Theta,\Delta \quad \text{if } r \in \text{dom}(\Theta) \quad (\text{READ}) \]
\[ E[\text{writeTV }r M];\Theta,\Delta \Rightarrow E[\text{return }\Theta];\Theta[r \mapsto M],\Delta \quad \text{if } r \in \text{dom}(\Theta) \quad (\text{WRITE}) \]
\[ E[\text{newVar }M];\Theta,\Delta \Rightarrow E[\text{return }r];\Theta[r \mapsto M],\Delta \quad \text{if } r \notin \text{dom}(\Theta) \quad (\text{NEW}) \]

\[ M \xrightarrow{\tau} N \quad (\text{ADMIN}) \]
\[ E[M];\Theta,\Delta \Rightarrow E[N];\Theta,\Delta \quad E[M_1];\Theta,\Delta \Rightarrow E[M_1';\theta,\Delta'] \quad (\text{OR1}) \]
\[ E[M_1];\Theta,\Delta \Rightarrow E[\text{throw }N];\Theta',\Delta' \quad E[M_1];\Theta,\Delta \Rightarrow E[\text{throw }N';\Theta',\Delta'] \quad (\text{OR2}) \]
\[ E[M_1];\Theta,\Delta \Rightarrow E[\text{retry}];\Theta',\Delta' \quad E[M_1];\Theta,\Delta \Rightarrow E[M_2];\Theta,\Delta \quad E[M_1];\Theta,\Delta \Rightarrow E[M_2];\Theta,\Delta \quad (\text{OR3}) \]

Figure 4: Operational semantics of STM Haskell

What if \( n \) 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), (OR2), and (OR3). 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 \( \Theta' \). 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 \( M_2 \).

An alternative choice for (OR2) would have been to 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

\[ M \text{ 'orElse' retry } = M \]

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!

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.

The Haskell runtime currently operates only on a uniprocessor, scheduling lightweight Haskell threads within a single operating system thread. Haskell threads are only suspended at well-defined “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 our uniprocessor design [8, 7]. 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 multi-processor 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 TVars 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 memory transaction, all TVar accesses are performed by STMReadTVar and STMWriteTVar (Figure 5). These accesses 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 TLogs and TVars. It depicts a memory 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 the current value from the TVar, sees it to be Nothing and writes the value Just 42 in its place, indicating that the number 42 is to be placed in the buffer.

6.2 Validation and commit

The atomic function is implemented by pushing an AtomicFrame record onto the Haskell execution stack, and invoking STMStart to allocate a fresh transaction log (Figure 5). When execution returns to the AtomicFrame, 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 (rule ATTHROW of 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:

\[
\begin{align*}
  f &:: \text{Integer} \rightarrow \text{Bool} \\
  f x &= \text{if } x==0 \text{ then True else } f (n-1)
\end{align*}
\]

If \( f \) 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. Every time the scheduler is about to devote a time-slice to running 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 than doomed transactions can be killed off before they have consumed too much time. It does not make sense to validate more frequently than doomed transactions can be killed off before they have consumed too much time.

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, which link off the TVars that the transaction has accessed, 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 TVars: 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 and blocks once more – this can happen
only if, by the time the thread is scheduled, the TVar again contain pointer-equal values to those originally read by that thread.

Figure 7 depicts two threads both waiting for updates to be committed to the same TVar. In this case the two threads both saw Nothing within the TVar.

### 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, write records are (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 STMInvalid is called as usual to check that the nested transaction 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 invoke retry then we need to propagate the retry in such a way that the thread will wait on the union on the sets of TVars that they have accessed. To do this, we first call STMIsValid to validate the logs for both nested transactions. If either is invalid then we re-execute the atomic function with a fresh log. Otherwise, if both nested logs 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 TVars. 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 TVars 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:

```haskell
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 passed to the final call of STMMergeNested. However, when retrying, the new value in the merged log does not matter: while the retry is propagating the log will be subject only to further merges, or eventually to STMMerge.

### 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.

Figure 8: Two steps in the implementation of 'orElse'

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.

### 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 in STM Haskell, which mirrors the channel abstraction of Concurrent Haskell [20] implemented using MVars. 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. 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 [15]. 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 to provide fault-tolerance in databases [5] and distributed systems. These transactions rely on stable storage and distributed commit protocols to protect system integrity against crashes and communication failures.

Nested transactions were first proposed by Moss [17], who extended nesting to two-phase locking protocols. The Argus language [14] for fault-tolerant distributed applications provided explicit language support for nested transactions.

Distributed transactions typically provide both synchronization, 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 synchronization for transactions running at a single machine. These projects include RVM [24], ObjectStore [13], and the PJama extensions to Java [11].

By contrast, software transactional memory provides synchronization without persistence. Because the state manipulated by memory transactions is not intended to survive crashes or communication failures, 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 [9, 26] to support non-blocking synchronization, and architectural support for this model remains the subject of ongoing research [6, 16, 18, 22]. A number of proposals have emerged for supporting transactional memory in software [7, 8, 10, 25].

Work on software transactional memory has focused on libraries, not on integrating transactional mechanisms into a programming language. Two exceptions are Welch et al. [27] who show how STM-like techniques can increase the concurrency available in systems based on Java’s synchronized blocks, and Harris and Fraser [7] who discuss how Java might be adapted to support non-blocking atomic sections. This prior work did not place much emphasis on mechanisms for conditional blocking or compositionality. Herlihy et al. [8] support syntactically nested transactions by “flattening” nested transactions to a single transaction, but provide no explicit mechanism for conditional blocking. Harris and Fraser [7] 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 [23] is an inspiring language directed squarely at the goal of composable concurrency. The principal abstraction is that of a first-class event, which allow far richer composition than do conventional locks, or Concurrent Haskell’s MVars. 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:

```
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, with type

```
swap :: SwapChan a -> a -> STM a
```

because that would require mutual linkage of an arbitrary number of transactions 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 has an optimistic-concurrency mechanism called proposals that supports a subset of our notion of a memory transaction [12]. Each thread maintains a proposal (= transaction log) which logs reads and writes, provided the latter are performed using the logging operations provisional-car, provisional-set-car!, etc. The call call-ensuring-atomicity t is just like our atomic t; it retries automatically if t sees an inconsistent view of memory.

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 transactional memory provides a substrate for concurrent programming that offers very much richer composition than has been available to date, and that it can be implemented in a practical programming 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 \texttt{retry} and \texttt{orElse} can indeed be applied to other languages. For instance, in C\#8, one could indicate \texttt{retry} by raising a specified kind of exception and then express \texttt{orElse} as a particular kind of exception handler.

In a mainstream object-oriented 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 in CVS, and will shortly be released in binary form (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, Matthew Flatt, Jim Gray, Dan Grossman, Andres Löh, Jon Howell, Jayadev Misra, Norman Ramsey, Michael Ringenburg, David Tarditi, and especially Tony Hoare, for their helpful feedback on this paper. A special thank-you to Andres Löh for help with typesetting the figures.

References


