Modular Transactional Memory

Matthew Brecknell  
matthew@brecknell.net  
Queensland University of Technology, Australia

Paul Roe  
p.roe@qut.edu.au

Abstract

Software transactional memory has the potential to greatly simplify development of concurrent software, by supporting safe composition of concurrent shared-state abstractions. However, STM semantics are defined in terms of low-level read and write operations on individual memory locations, so implementations are unable to take advantage of the properties of user-defined abstractions. Consequently, the performance of transactions over some structures can be disappointing.

We present Modular Transactional Memory, our framework which allows programmers to extend STM with concurrency control algorithms tailored to the data structures they use in concurrent programs. We describe our implementation in Concurrent Haskell, and two example structures: a finite map which allows concurrent transactions to operate on disjoint sets of keys, and a non-deterministic channel which supports concurrent sources and sinks.

Our approach is based on previous work by others on boosted and open-nested transactions, with one significant development: transactions are given types which denote the concurrency control algorithms they employ. Typed transactions offer a higher level of assurance for programmers reusing transactional code, and allow more flexible abstract concurrency control.

Keywords: Software transactional memory, concurrency control, transactional boosting, open-nested transactions.

1 Introduction

Software transactional memory (Shavit & Touitou 1997) has emerged as a promising alternative to lock-based techniques for controlling concurrent access to shared data structures. STM provides an abstraction of physical memory, in which arbitrary sequences of memory operations can be executed as transactions, with the expected semantics of atomicity, consistency and isolation. STM saves the programmer from many of the pitfalls of lock-based programming, such as race conditions (failing to take the correct locks) and deadlocks (taking locks in the wrong order). Importantly, STM transactions are composable (Harris et al. 2005), whereas structures built using locks and condition variables typically are not. The latter has major benefits for the modularity and reusability of concurrent software.

Yet, if we care about performance, composability can be illusory. The semantics of transactional memory are typically defined by whether read and write operations on individual memory variables commute. While this is sufficient to ensure correctness under composition, it is not always necessary. Abstractions already limit the admissible sequences of memory operations, often in ways that would allow more efficient concurrency control. STM implementations are unable to take advantage of this fact, so the performance of concurrent abstractions can be worse than one might intuitively expect.

Transactional boosting (Herlihy & Koskinen 2008) and open-nesting (Ni et al. 2007) are methodologies which address this issue, allowing programmers to improve concurrency without losing composability. Boosting allows existing data structures to be used in transactional contexts, and open-nesting allows such data structures to be constructed using STM. Both allow programmers to replace low-level concurrency control on individual memory locations with higher-level abstract concurrency control.

Previous implementations providing boosted and open-nested transactions allow execution of arbitrary code in transactional contexts. This is necessary for abstract concurrency control, but it weakens the STM abstraction. The use of untyped transactions also limits the expressivity of abstract concurrency control, resulting in suboptimal implementations. In this paper, we develop a more disciplined approach, making the following contributions:

- We propose types for modular transactional concurrency control. Transactions are typed according to the concurrency control algorithms they employ, for improved assurance and flexibility of abstract concurrency control.

- We describe our implementation in Concurrent Haskell, combining transactional boosting and open-nested transactions with modular blocking and choice.

- Since our system uses a two-phase commit protocol internally, we provide a simple two-phase commit mechanism to users at no extra cost.

This paper is organised as follows. After some critical background (§2), we give an overview of our system (§3), some example applications (§4), and a discussion of future directions (§5). Finally, we draw conclusions (§6).

2 Background and related work

To make the work accessible to a general audience, we include the following background material. We introduce software transactional memory (§2.1), and show
how transactional boosting and open-nested transactions can be used to improve concurrency without losing composability. We also provide a brief overview of Concurrent Haskell (§2.2), since our system makes crucial use of certain features of the language.

2.1 Software transactional memory

STM is an abstraction of shared memory, in which arbitrary sequences of memory operations are executed as transactions, such that concurrent transactions appear to execute serially. Programmers can use familiar sequential reasoning within transactions, and only need to ensure that transactions take consistent states to consistent states.

For example, in a transaction to debit an amount, we can be sure that concurrent updates will not modify the balance between the times we read and write the balance. In pseudo-code:

```java
atomic debit(amount) {
    balance := balance - amount;
}
```

Although this appears like a critical section synchronised on the account, note that critical sections often do not compose correctly. For example, to atomically debit multiple accounts, it would be necessary to lock all accounts before any can be modified.

Transactions do compose naturally, and concurrent processes cannot observe intermediate states. For example, during the following transfer, no other process can observe a state where the amount is absent from both accounts:

```java
atomic transfer(amount, debtor, creditor) {
    debtor.debit(amount);
    creditor.debit(-amount);
}
```

An atomic method may contain primitive operations (reads and writes) on transactional memory, as well as calls to pure functions and other atomic methods. An atomic method called from an ordinary method is executed as a top-level transaction, while an atomic method called from another atomic method is executed in the context of the calling transaction.

There is no need to specify locks; the runtime system determines which locks are required by observing the memory locations accessed during execution. No deadlock is possible, even when there are concurrent transfers in opposite directions; the system aborts and restarts transactions as necessary to ensure isolation and global progress.

2.1.1 Informal semantics

STM ensures opacity\(^1\) of concurrent execution traces (Guerraoui & Kapalla 2008). Opacity is similar to the serialisability criteria used to describe database transactions, but also requires that even transactions which abort only observe consistent states. Informally, an execution trace is opaque, or serialisable, if it is equivalent, in a certain sense, to some sequential execution of the same transactions.

Equivalence of execution traces can be defined in terms of conflicting operations. Concurrent operations conflict when they cannot be reordered without potentially changing the outcome of some subsequent operation. For simple read-write memory variables, a write conflicts with any other operation on the same variable, irrespective of the value written. This provides a simple abstraction, but it can result in disappointing performance. Consider a simple integer map:

```java
class map_stm {
    atomic int lookup(int key);
    atomic insert(int key, int value);
    atomic delete(int key);
}
```

In an abstract sense, operations on distinct keys can be reordered without affecting subsequent operations. However, an STM implementation of this structure is likely to generate conflicts between operations on distinct keys. This can easily be seen in a sorted linked list (figure 1), where deletion of the first node requires a write to the head, which necessarily conflicts with any other operation on the structure. Similar conflicts occur even in more sophisticated structures, such as hash tables and skip lists.

At the level of the operations themselves, such conflicts are unavoidable. The problem is that the impact of these conflicts on concurrency may leak far beyond the scope of those operations. For example:

```java
atomic update(int key, map_stm src) {
    value := src.lookup(key);
    new_value := expensive_computation(key, value);
    src.insert(key, new_value);
}
```

Conflicts generated internally by map_stm operations mean that update operations on distinct keys might not be able to execute concurrently. That is, update operations are unnecessarily serialised, including any unrelated but expensive computations they contain.

To rectify this loss of composability, the notion of conflict must be generalised to include the operations of user-defined abstractions, for abstract concurrency control. The following sections describe two existing approaches.

2.1.3 Transactional boosting

Transactional boosting (Herlihy & Koskinen 2008) allows existing non-transactional data structures to be used in transactional contexts; for example, the containers in java.util.concurrent. Boosting is possible for structures where each operation has an inverse.

\(^1\)Most existing transactional memory implementations ensure opacity (Guerraoui et al. 2008), so opacity might be considered the de-facto standard semantics for STM.

\(^2\)Opacity actually requires a slightly stronger notion of equivalence, but we find the notion of conflict useful for the discussion of abstract concurrency control which follows.

Figure 1: Deletion from the front of a linked list
and is linearisable\(^3\), and where the commutativity of operations can be identified. Boosting makes use of compensating actions to undo the effects of operations when a transaction must abort, and abstract locks to ensure that non-commutative operations occur in a serialisable (opaque) order.

Our pseudo-code language is extended with new keywords, \texttt{boosted} and \texttt{onAbort}. A \texttt{boosted} method may be called from an atomic method, and contains non-transactional code to access the boosted structure and acquire abstract locks. Within a \texttt{boosted} method, \texttt{onAbort} may be used to record compensating actions to execute if the transaction is aborted. Abstract locks are acquired in a lock manager which holds locks until the acquiring transaction commits or aborts, and which aborts transactions as necessary to avoid deadlock.

Given an integer-indexed lock structure \texttt{lock_idx}, an existing linearisable integer map may be boosted as follows:

```haskell
class map_boosted {
    map_linear map;
    lock_idx lock;
    /
    boosted delete(int key) {
        lock.shared(key);
        value = map.lookup(key);
        if (value is not null) {
            lock.exclusive(key);
            onAbort () {
                map.insert(key, value); }
            map.delete(key);
        }
    }
}
```

Operations on the underlying structure are already linearisable, and so do not need any further synchronisation. However, locks are required to ensure abstract serialisability among boosted transactions. Since operations on distinct keys commute, one lock per key is sufficient for this purpose. Further, shared locks can be used for read-only operations.

The benefit of boosting this structure is that operations like \texttt{update} (\S\ref{sec:serial}) can reach a higher degree of concurrency. Using existing highly concurrent data structures, the concurrency between individual map operations can be significantly improved. More importantly, abstract locking may also allow other parts of transactions to run with improved concurrency. If those other parts are expensive, the improvement can be dramatic.

In our framework \cite{PeytonJones96}, typed transactions provide the same benefits as transactional boosting, but with greater flexibility. For example, typed transactions allow individual compensating actions to be aggregated into more efficient bulk operations.

### 2.1.4 Open-nested transactions

Open nested transactions \cite{Ni07} are nested transactions which commit as soon as they complete, making their effects visible to other processes before their parent transactions commit. Together with abstract locking, and compensating actions in case the parent transaction aborts, open-nested transactions provide similar benefits to transactional boosting.

In full generality, open-nested transactions do not have clear semantics. For example, there is no reasonable answer to the question of how an implementation should behave when an open-nested transaction and its parent both modify the same memory location. We therefore consider a restricted usage, in which open-nested transactions are only used to implement abstractions in which:

- every operation which accesses internal state of the abstraction is an open-nested transaction,
- those operations only access the internal state of the abstraction, and
- internal state is encapsulated, and is not directly accessible from outside the abstraction.

Agrawal et al. \cite{Agrawal08} give an ownership type system which formalises and enforces a similar, but more refined restriction.

The use of open-nested transactions is then very similar to transactional boosting, except that the underlying structure is implemented using STM transactions. One difference is that open-nested transactions may contain open-nested transactions as descendants, whereas boosted transactions cannot be nested.

### 2.2 Concurrent Haskell

We have developed our system in Concurrent Haskell \cite{PeytonJones96}, as it provides a useful laboratory for experimenting with new ways of writing imperative programs, primarily because side-effects are associated with types. For readers unfamiliar with the language, we provide a brief overview here.

Haskell is a purely functional programming language, in which all values are immutable and all functions are free of side effects. Concurrent Haskell provides explicit threads and synchronisation as an extension provided by the Glasgow Haskell Compiler.

Useful programs must have side effects, so Haskell contains an imperative sublanguage which is embedded as an abstract parametric datatype, \texttt{IO}. Values of \texttt{IO} types describe actions which may be executed. If executed, an action may have side-effects, and may return a result to its calling action. For example, following are the type signatures for two members of the standard library:

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

These are read as follows: \texttt{getChar} is an action which, if executed, may have side-effects (in this case, reading a character from the terminal) and returns a character result; and \texttt{putChar} is a function which takes a character as its argument and returns an action which, when executed, may have side-effects (printing the character) and returns void.

Actions may be composed using various combinators, or alternatively, using do-notation:

```haskell
echo :: IO ()
echo = do { c <- getChar; putChar c; echo }
```

Actions are first-class values, like those of any other type. Purity demands that the mere construction or manipulation of \texttt{IO} actions does not cause them to be executed. Indeed, an action is only executed if it is the single distinguished \texttt{IO} action called \texttt{main}, or one of its composite parts.

Since \texttt{IO} can describe arbitrary side-effects, it can also describe mutable memory cells, with type \texttt{IORef}. Manipulation of mutable cells only occurs within \texttt{IO} actions, so there is no contradiction of purity. In these signatures, the identifier \texttt{a} is a type variable, indicating that these functions, and \texttt{IORef}, have parametric polymorphic types.
Deterministic choice is provided by orElse, which takes two transactions, and runs the first, unless it would block with a call to retry, in which case it runs the second. For example, the following will debit the first of two accounts to have clear funds:

```haskell
debitFirst :: Int -> TVar Int -> TVar Int -> STM ()
debitFirst act1 act2 = debitWhenClear act1 'orElse' debitWhenClear act1 act2
```

Note one of the benefits of a pure type system: transactions and IO actions have distinct types, so the type rules ensure that transactions only contain operations which can be performed atomically. Our work aims to realise similar benefits for abstract concurrency control.

2.2.2 Type classes

Haskell type classes\(^4\) provide a disciplined approach to overloading. For example, following is a type class for equality comparison:

```haskell
class Eq a where
  (==) :: a -> a -> Bool
```

This defines a new type class Eq with an overloaded operator method (==). The type variable (a) stands for each type which will instantiate the class, and must appear somewhere in the type of each method.

A type instantiates a class with an instance declaration which provides a suitable implementation for each method, specific to that type. For example, Bool equality can be defined by pattern-matching on the constructors of the type:

```haskell
instance Eq Bool where
  True == True = True
  False == False = True
  _ == _ = False
```

The Eq type class allows functions which work for all types supporting equality. For example, to test if an item is contained in a cons-list, by pattern-matching on the constructors for the list type:

```haskell
elem :: Eq a => a -> [a] -> Bool
```

Here, the first part of the type signature (Eq a =>) indicates that the type variable (a) is constrained. That is, elem is only defined for types supporting equality. Square brackets around a type ([a]) indicate a list of that type; a single colon (x:xs) constructs a cons-cell which preprends an item to an existing list; and empty square brackets ([]) construct an empty list.

Instances may be defined in terms of simpler instances. For example, equality for lists is defined in terms of equality of elements:

```haskell
instance Eq a => Eq [a] where
  (x:xs) == (y:ys) = (x == y) && (xs == ys)
  [] == [] = True
  _ == _ = False
```

This enables a form of type-directed programming.

3 Modular transactional memory

This section introduces our system. The design is inspired by Haskell STM, with support for abstract concurrency control. Transactions are attributed with descriptive types, to reconcile the need for arbitrary side-effects during abstract concurrency control, with the benefits of static guarantees provided by a pure type system. Typed transactions can also improve the expressivity and efficiency of abstract concurrency control.

3.1 Typed transactions

We begin by describing transactions and how they can be composed, without reference to any particular transactional data structures. A new type parameter is added to the type of transactions, to represent the concurrency control algorithms used, giving the type (Tx t a) of transactions using algorithms described by type t and returning type a.

In this context, an algorithm is the concurrency control (locking, compensating actions, etc.) for some abstract structure. For example, if Mem (defined elsewhere) is the type representing the concurrency control for ordinary transactional variables, then we can

\(^4\)Haskell type classes are almost entirely unlike the classes of object-oriented programming. In particular, they are not types.
form the type \( (Tx \text{ Mem } a) \) of transactions over ordinary variables. Indeed, the latter corresponds to the type \( \text{STM} \) in the existing Haskell STM.

\( Tx \) is defined as an IO action, which executes in the context of a central transaction log of type \( \log \) (§3.3), and a local transaction log of type \( t \):

```haskell
data Result a = Abort | Retry | Result a

newtype Tx t a =
  Tx { runTx :: Log -> t -> IO (Result a) }
```

The Result type is used internally to distinguish between transactions which abort and restart due to a conflict, those which call retry, and those which return a result normally.

Thus, if concurrency control algorithms are types, then their values are local transaction logs, specialised to process transactions over the corresponding data structures. For composition of transactions, a transaction log must provide certain operations, including initialisation, nesting and finalisation. These requirements are represented as a type-class:

```haskell
class TLog t where
  run :: (t -> Undo -> Commit -> IO a) -> IO a
  nest :: t -> (Undo -> IO a) -> IO a
```

\( TLog \) methods are called by the framework, which passes an IO action as a callback. A method implementation should execute the callback, passing the required parameters back to the framework. Typically, method implementations will also wrap the callback with resource management and exception handlers.

The type Commit describes the actions to perform when a transaction commits, and will be explained later (§3.4). The type Undo describes an action which reverses the effects of a transaction when it aborts, and is a simple type alias:

```haskell
type Undo = IO ()
```

The framework calls run to begin execution of a transaction. The implementation of run must initialise a new transaction log (type \( t \)), as well as global abort and commit actions (types Undo and Commit) to pass back to the body. The framework then calls nest to obtain a local undo action for every call to catch and orElse.

An instance of TLog is sufficient to implement all of the core combinator of Haskell STM, including overloading the do-notation, and these functions:

```haskell
catch :: (TLog t, Exception e) -> Tx t a -> (e -> Tx t a) -> Tx t a
retry :: TLog t -> Tx t a
orElse :: TLog t -> Tx t a -> Tx t a
atomically :: TLog t -> Tx t a -> IO a
```

For example, the following shows the implementation of catch, which presents nested transactions as an exception-handling mechanism. Here, backslash (\) introduces a lambda (anonymous function):

```haskell
catch body handler = Tx (\tl t -> nest t (\undo ->
  Control.Exception.catch (runTx body l t)
  (\e -> do { undo; runTx (handler e) l t }) )))
```

Haskell programmers might be alarmed at the pervasive use of IO in the definitions of TLog and Tx. Do we fail to take advantage of the benefits of a pure type system? We do not believe so, because every use of IO in the definition of a transactional data structure is labelled with the type \( t \) of the local transaction log. Thus, transaction types act as certificates of origin, which can help a user to understand the nature of some transactional code, without the need to study all of its parts.

### 3.2 Combining transaction types

The real value of this framework is in its support for composing transactions over arbitrary combinations of data structures, with abstract concurrency control. Since we can only compose transactions of the same local log type, we provide a type-level operator (:&:) to construct combined local log types, and a mechanism to inject transactions into these combined types. For example, if Map is the type representing concurrency control for associative maps, then we can form the type \( (Tx \text{ Mem } :&\text{ Map } a) \) of transactions over ordinary variables and associative maps. The operator is implemented as a simple right-associative pair type, which may be right-nested to arbitrary depth:

```haskell
data l :& r = 1 :& r
infixr 5 :&
```

It is straightforward to provide a TLog instance for a combined type, based on TLog instances for the component types:

```haskell
instance (TLog l, TLog r) => TLog (l :& r)
```

It is also easy to implement the following explicit injections, which could be used to compose two transactions over different structures, in sequence:

```haskell
inj1 :: Tx l a -> Tx (1 :& r) a
inj2 :: Tx r a -> Tx (1 :& r) a
```

However, explicit injection functions are cumbersome to use, so we define an inclusion relation which automates injections. This is implemented as a multi-parameter type class (:<) with suitable instances for combined types, following Swierstra (2008):

```haskell
class t :< c where
  inject :: Tx t a -> Tx c a
```

For example, given transactions over Mem and Map, with the following types:

```haskell
example_mem_basic :: Tx Mem ()
example_map_basic :: Tx Map ()
```

We can inject into an unspecified, but constrained, combined transaction type, resulting in transactions with the following types:

```haskell
example_mem :: (Mem :< c) => Tx c ()
example_mem = inject example_mem_basic
example_map :: (Map :< c) => Tx c ()
example_map = inject example_map_basic
```

These can then be composed in sequence, resulting in a transaction with the following type:

```haskell
example :: (Mem :< c, Map :< c) => Tx c ()
example = do { example_mem; example_map }
```

The type can be read: example is a transaction over any set of structure types containing at least Mem and Map. This example transaction could be further composed with other transactions, with new constraints added to the transaction type as necessary.

Finally, at the point of execution, the transaction must be given a concrete type, to complete the automatic injection:

```haskell
example_exec =
  atomically (example :: Tx (Mem :& Map) O)
```

By combining abstractions this way, we assume that the areas of memory accessed by those abstractions are non-overlapping. This is a reasonable assumption, provided abstractions are properly encapsulated. It is also consistent with the restriction on the use of open-nested transactions (§2.1.4).
3.3 Primitive transactions

So far, we have described transactions and how they compose in a very general sense, without saying anything about how they are implemented. In particular, we have not shown how to provide primitive transactions for any structure of interest.

We are forced to make some significant implementation choices. There are numerous approaches to STM implementation, with equally many trade-offs (Larus & Rajwar 2007). Many approaches suit certain workloads, but not others. Since we are still in the early stages of developing this framework, our choices have been determined primarily by the need for simplicity, and secondly by a bias toward low transactions and high-contention workloads, since we expect this to be the area where abstract concurrency control will provide the greatest benefit. We intend to revisit these choices as we continue development.

We have opted to implement transactions with visible readers. Whenever a transaction reads state from a data structure, it must register with that structure to receive notifications when updates by other transactions invalidate that observation. For serialisability, we abort any transaction which receives such a notification before it is ready to commit. Processor cache performance can be inhibited by visible readers, but we currently accept this for the sake of simplicity.

The system allows both deferred and direct updates, although support for direct updates is limited. In a deferred update, a transaction records its intention to update to its local transaction log, but the actual update is deferred until the transaction commits. Deferred updates only require locks to be acquired during the commit process, but read operations must consult the transaction log before the data structure in memory. During a direct update, a transaction immediately acquires locks on the structure, records the previous state in its transaction log (in case the transaction aborts), and writes the update directly to the structure. Transactions using direct updates must take care to avoid deadlock, usually by aborting the transaction if a lock cannot be acquired within a short time limit. Currently, there is no support for deadlock detection, lock pre-emption or cascading aborts for direct updates.

A primitive transaction for a structure is created using the primitive function, which has the following type:

\[
\text{primitive} :: \text{Cert} t \rightarrow (\text{Log} \rightarrow t \rightarrow \text{IO (Result a)}) \rightarrow \text{Tx} t a
\]

Cert is a type family used to restrict the construction of a primitive transaction to the module which defines the corresponding structure. Constructors for values of type \((\text{Cert} t)\) should be private to the module which defines type \(t\). This parameter thus certifies that the calling module is authorised to define a transaction of the given type.

The second argument provides the implementation for the primitive transaction. It is an IO action with access to the central transaction log (type Log) and the local transaction log (type t) for the respective structure type, and which returns a Result type. This means that a primitive transaction may return any ordinary result, or alternatively, Abort or Retry. Returning Abort causes the transaction to abort, for example, if a lock for a direct-mode update could not be acquired. Retry indicates that the primitive transaction should behave like a call to retry.

The local transaction log (of type t) is defined entirely by the module providing that transaction type. Typically, it records updates, in either deferred or direct mode. Examples given later (§3.4) show how local transaction logs can improve the expressiveness of abstract concurrency control.

The central transaction log (type Log) is used to implement the visible reader protocol. A transaction which observes the state of a structure should register its Log with that structure. Concurrent transactions which update the structure must notify any registered transactions which are invalidated by those updates, by calling invalidate, which has type:

\[
\text{invalidate} :: \text{Log} \rightarrow \text{IO ()}
\]

To maximise the benefit of abstract concurrency control, the implementation of a transactional data structure should aim to send notifications only to transactions whose observations have in fact been invalidated, according to the abstract semantics of the structure.

For transactions over multiple structure types, it is important that invalidation is handled centrally, since an update to any previously observed structure could invalidate the transaction. The implementation of primitive checks that the transaction has not been invalidated after executing the body of each primitive transaction.

Conveniently, this mechanism also serves to implement conditional blocking. A transaction which calls retry simply waits for invalidation, and then aborts and restarts.

3.4 Commit actions

As a transaction executes, each local log accumulates updates in deferred or direct mode. When finished, it must either commit or abort. It can only commit if it can acquire suitable locks for deferred updates, and if it has not been invalidated by an update performed by another transaction.

The commit process is straightforward. Deferred-mode updates can simply be discarded. Direct-mode updates must be reverted, but this can be done safely, since locks have already been acquired. Locks must then be released, and reader registrations should be cleared.

The commit process is more complicated, and proceeds in the following phases:

1. Locks must be acquired for all deferred updates across all local transaction logs. If it is not possible to acquire all locks, then the transaction must abort without any updates being committed.
2. The transaction must be validated, with a check that the transaction has not been invalidated by updates committed by another transaction. Otherwise, it must abort.
3. Only if the transaction has not aborted, then deferred updates must be written across all local transaction logs, and notifications sent to invalidated transactions.
4. In any case, locks must be released, and reader registrations cleared.

We refer to the first and third phases as the prepare and update phases, respectively. They are performed by local transaction logs, while the second (validation) phase is performed centrally.

The system collects actions to execute for the prepare and update phases from local transaction logs, via the run method of the TLog class (§3.1). The actions are described by the Commit data type:

\[
\text{data Commit} = \text{Commit} (\text{prepare} :: \text{IO ()} \rightarrow \text{IO ()}, \text{update} :: \text{IO ()})
\]

18
To begin the commit process, the framework constructs a chain of calls to `prepare` and `update` for all local logs involved in the transaction, such that each call to `prepare` is passed a callback consisting of any remaining calls to `prepare`, followed by validation and a sequence of `update` actions. Each `prepare` action should execute its callback only if it successfully acquires all locks necessary to ensure that the corresponding `update` can execute correctly. When the callback returns, the transaction has either aborted or committed, so `prepare` may release locks it previously acquired.

### 3.5 User-level two-phase commit

With the internal use of two-phase commit, it is possible to provide a form of two-phase commit directly to the user, without any additional cost. We provide `twophase`, with the following type:

```haskell
  twophase :: TLog t => Tx t (IO a) -> IO a
```

This is similar to `atomically`, but instead of taking a transaction returning an ordinary value, `twophase` takes a transaction returning an `IO` action. We refer to this as the `validate` action.

When executing a transaction, `twophase` performs the validate action between the prepare and update phases of the commit protocol. This guarantees that the transaction can commit, but there is still the option to abort before updates become visible to other transactions.

We define the operation of `twophase` as follows: if the validate action returns normally, then the transaction commits, and `twophase` returns the same result as the validate action; if the validate action throws an exception, then the transaction aborts, and `twophase` propagates the exception.

For example, the following requires confirmation from the user before debiting an account:

```haskell
  debitConf account = twophase (do
    balance <- readTVar account
    writeTVar account (balance - amount)
    return (do
      confirm <- prompt "Balance = " ++ show balance
      if not confirm
        then throw RejectDebit
        else return ()))
```

Without `twophase`, it would not be possible to ensure that the balance shown to the user is the same balance to which the debit is applied.

### 3.6 Blocking open-nested transactions

Open-nested transactions require special treatment, if they are to be allowed to include calls to `retry`. Specifically, if an open-nested transaction blocks with a call to `retry`, and the parent transaction is invalidated by an update committed by another transaction, then the open-nested transaction should return control to the parent, to allow it to restart. The converse is not necessary: if an open-nested transaction is directly invalidated, but its parent has not been invalidated, then only the open-nested transaction need restart.

Further, if an open-nested transaction calls `retry` when the parent transaction is in the first branch of the choice operator (`orElse`), then the open-nested transaction should not block, but should immediately abort and return control to the parent, to allow it to execute the alternate `orElse` branch.

These requirements are satisfied by `open`, which has the following type:

```haskell
  open :: TLog t => Log -> Tx t a -> IO (Result a)
```

Like `atomically`, `open` executes a transaction, but requires an additional parameter for the parent transaction `Log`. The implementation of `open` uses this to register with the parent, to receive its invalidation signals, and to determine whether the parent is in the first branch of `orElse`. An open-nested transaction will only block if the parent is not in the first branch of `orElse`.

A call to `open` returns a `Result` type. `Retry` is returned when the parent is in the first branch of `orElse`, and the open-nested transaction calls `retry`. `Abort` is returned when the parent transaction has been invalidated. This result should be returned to the parent transaction (via `primitive`), after performing appropriate abstract concurrency control.

### 4 Example structures

This section describes several example transactional structures. First, we develop the prototypical structure, the simple transactional variable, and then contrast this with a boosted associative map structure. Finally, we develop a non-deterministic channel using open-nested transactions. Readers should bear in mind that the combining operators provided by the framework (§3.2) allow transactions over these structures to be composed together.

#### 4.1 Simple transactional variables

In this framework, simple transactional variables are defined as any other structure. We define the structure itself, its local transaction log, and its primitive transactions. Using deferred-mode updates, the `TVar` structure may be defined as follows:

```haskell
  data TVar a = TVar { tvar_data :: IORef a, tvar_lock :: MVar (), tvar_readers :: CList Log, }
```

Recalling that `IORef` is a mutable variable, and `MVar` provides synchronisation, the purpose of the first two fields should be clear. The third registers transactions which have observed the value of the `TVar`, for notification of updates. `CList` is a list type which supports concurrent insertion and deletion, and is necessary because many transactions may attempt concurrent reads from a `TVar`. The local transaction log can then be defined as follows:

```haskell
  data Write = forall a. Write (Tvar a) a
  data Mem = Mem { mem_read :: IORef [CNode Log], mem_write :: IORef [Write], mem_read :: IORef [CNode Log] }
```

The `mem_write` field records write operations in deferred mode, and is used during subsequent read operations, as well as the commit process. Each record includes the `TVar` written to, and the value written. The `mem_read` field records reader registrations for all variables read during the transaction, and is used to clear those registrations at the end of the transaction. `A CNode` is simply a handle to a `CList` node.

The local transaction log, `Mem`, must be made an instance of `TLog`, so we must provide implementations for `run` and `nest`. Of these, `nest` is the most interesting:

```haskell
  nest mem body = do
    savedWrites <- readIORef (mem_write mem)
    body (writeIORef (mem_write mem) savedWrites)
```

19
For a nested transaction, the framework calls `nest` with the local transaction log (mem) and the body of the nested transaction. First, `nest` saves a copy of the write log, and then calls the body, passing an undo action which may be used to restore the saved write log in the event that the nested transaction aborts. Recall that constructing an 10 action does not cause it to be executed immediately. If needed, the framework will execute the undo action at the appropriate time; otherwise it will be discarded.

Implementations of primitive transactions must certify their authority to create a transaction of the required type. A private type definition, declared as a `Cert` instance, serves this purpose:

```haskell
data MemCert = MC

type instance Cert Mem = MemCert
```

The `readTVar` primitive may now be defined:

```haskell
readTVar :: TVar a -> Tx Mem a
readTVar var = primitive MC (log t -> do
  look_aside <- write_log_lookup var t
  case look_aside_of
  Just value -> return (Result value)
  Nothing -> readDirect log t var
)
```

In case the transaction has previously performed an uncommitted write to the same variable, a read operation first examines the write log. If the `TVar` is not found, `readDirect` is called to insert the transaction log into the `tvar_readers` field of the `TVar`, add the registration to the `mem_read` field of the local transaction log, and return the current value of the variable. To ensure reads are properly sequenced with respect to writes in other transactions, `readDirect` takes a lock on the `TVar`.

The `writeTVar` primitive simply adds a `Write` entry to the `mem_write` field of the local transaction log, or creates an existing entry for the same `TVar`. Then, at commit time, the `prepare` action attempts to acquire locks for all `TVars` in the write log:

```haskell
prepare callback = do
  write_log <- readIORef (mem_write t)
  foldr tryWithLock callback write_log
```

Here, `foldr` reduces the write log with `tryWithLock`, which takes a `Write` entry and a callback, and executes the callback only if it acquires a lock on the corresponding `TVar`. If `prepare` succeeds, `update` writes each new value in the write log to the corresponding `TVar`, and invalidates any transactions which have registered reads on each `TVar`. The abort action is empty, but the `run` method clears reader registrations which have accumulated in the `mem_read` field of the local transaction log, at the end of the transaction.

### 4.2 Finite map

The map structure presented here is the moral equivalent of the boosted map shown earlier (§2.1.3), since it consists of an underlying linearisable data structure wrapped with abstract concurrency control. However, there are substantial differences in our method: the typed local transaction log is accessible to read operations, and therefore supports deferred-mode updates; it also allows us to aggregate operations on each map structure, for a more efficient commit process.

In fact, this implementation is very similar to that of simple transactional variables, so we only show key differences. Unfortunately, there are no readily available concurrent map implementations for Haskell, so we use a purely functional map (shown as `M.Map` below) protected by a single lock. Although not concurrent, it is linearisable, and so still meets the requirements for boosting. Despite the absence of operation-level concurrency, abstract concurrency control is still valuable, since it may increase concurrency between other parts of transactions.

The `TMap` structure is defined as follows. The important difference is that `tmap_readers` is defined on a per-key basis, so that operations on distinct keys do not conflict.

```haskell
data TMap k v = TMap {
  tmap_data :: IORef (M.Map k v),
  tmap_lock :: MVar (),
  tmap_readers :: IORef (M.Map k (CList Log))
}
```

This complicates the process of clearing reader registrations, which must ensure that `tmap_readers` does not accumulate empty registration lists. The local transaction log therefore stores each reader registration with its associated key and `TMap`:

```haskell
data Write = forall k v.
  Write (TMap k v) (M.Map k (Maybe v))
data Read = forall k v.
  Read k (TMap k v) (CNode Log)
```

Note that each `Write` entry aggregates all the updates for a particular map, with deletions represented by a null value.

Primitive transactions, nesting and finalisation are similar to those for simple transactional variables, so we do not reproduce them here.

### 4.3 Non-deterministic unordered channel

This section describes an application of open-nested transactions, including those which block (§3.6), as well as direct-mode updates. More importantly, it also demonstrates the value of typed transactions to the consumer of transactions. To improve concurrency, we allow this structure to exhibit a higher level of non-determinism than is generally allowed by the semantics of closed word-based transactional memory. With this structure, different executions of the same transactions may yield different results, even if the serialisations are the same.

We believe that controlled non-determinism has a useful role in concurrent programming, because it can provide higher efficiency and concurrency. Nevertheless, it is a fundamental change in semantics, so transactions which may exhibit this behaviour should be clearly identified. Our framework achieves this by ensuring that every transaction which operates on this structure has a type which includes its local log type.

The `NDChan` structure, with local transaction log type `NDC`, is defined as an unordered, unbounded collection with operations `put` and `take`, where `put` inserts an item, and `take` either blocks or returns an item which it removes from the collection:

```haskell
put :: NDChan a -> a -> Tx NDC ()
take :: NDChan a -> Tx NDC a
```

The operation of `take` is non-deterministic, so it may choose any element from the collection, and may even block (retry) when the collection is not empty. We do want the structure to be useful, so we insist that `take` only blocks when there are concurrent transactions performing `take`, or the collection is empty.

Note that non-deterministic behaviour cannot be localised, but may infect the whole transaction. For example, since `take` may `retry` instead of returning an item, any `orElse` containing a `take` becomes a non-deterministic choice.
The underlying structure, TChan, is built with simple transactional variables. This is just the multiset channel described by Harris et al. (2005), with the important parts reproduced here:

```haskell
type Link a = TVar (Node a)
data Node a = Empty | Full a (Link a)
data TChan a = TChan { take_end :: TVar (Link a), put_end :: TVar (Link a) }
putTChan :: TChan a -> a -> Tx Mem ()
putTChan chan item = do 
  link <- readTVar (put_end chan)
  empty <- newTVar Empty
  writeTVar link (Full item empty)
  writeTVar (put_end chan) empty

takeTChan :: TChan a -> Tx Mem a
takeTChan chan = do
  link <- readTVar (take_end chan)
  node <- readTVar link
  case node of
    Empty -> retry
    Full item empty -> do
      writeTVar (take_end chan) next
      return item
While putTChan and takeTChan only conflict with each other if the channel is empty, each will always conflict with itself. This limits the concurrency between transactions performing the same operation on this structure, even if most of the work in those transactions is performed elsewhere. This problem is similar to the update operation described earlier (§2.1.2), and the solution is the same. While we cannot increase the concurrency between individual TChan operations, abstract concurrency control may allow other parts of transactions to run concurrently.

The non-deterministic specification of take affords a lot of freedom. We implement put in deferred-mode, and take as a direct-mode open-nested transaction. As abstract operations, put and take do not conflict with themselves, nor each other, with one exception: a put conflicts with a take which has blocked. This is the only situation in which we need to register a read operation. We do need to record items extracted by take, so that they may be returned if the transaction aborts. NDChan therefore contains a TChan, and a list of reader registrations:

data NDChan a = NDChan { ndchan :: TChan a, ndchan_read :: CList Log } The local transaction log (type NDC) records items extracted by take, deferred put operations, and reader registrations:

data Buffer = forall a. Buffer (NDChan a) [a]
data NDC = NDC { ndc_take :: IORef [Buffer], ndc_put :: IORef [Buffer], ndc_read :: IORef [CNode Log] }

Here, a Buffer is an association between a channel and a list of items of the same type.

The put primitive inserts a buffer to the ndc_put field of the local transaction log, or augments an existing buffer if the NDChan is already present.

The take primitive first searches the ndc_put field of the local transaction log. If take finds a buffer for the required NDChan, it removes and returns an item from that buffer. Otherwise, take executes an open-nested transaction to extract an item from the underlying TChan, and buffers the item in the ndc_take field of the local transaction log:

```haskell
let
  ndc :: NDChan a -> Tx NDC a
  ndc chan = primitive NDCert ($log t -> do
    look_aside <- extractPut chan (ndc_put t)
    case look_aside of
      Just item -> return (Result item)
      Nothing -> do
        result <- open log (takeTChan (ndcchan chan))
        case result of
          Abort -> return Abort
          Retry -> do
            node <- cinsert log (ndchan_read chan)
            modifyIORef (ndc_read t) (node:) return Retry
          Result item -> do
            buffer_take chan (ndc_take t) item
            return (Result item)
Note, we use atomically rather than open, since the abort action must not fail.
For commit, the prepare action is empty: it simply calls the body it is passed. The update action is similar to the abort action, inserting items buffered in ndc_take must be returned to their associated channels, and nodes registered on those channels must be invalidated:

```haskell
unbuffer :: Buffer -> IO ()
unbuffer (Buffer chan items) = do
  atomically (mapM_ unbuffer take_buffers)
  invalicate_clist (ndchan_read chan)

abort = do
  take_buffers <- readIORef (ndc_take t)
  mapM_ unbuffer take_buffers
```
```
Note, we use atomically rather than open, since the abort action must not fail.
finally, nest performs similar local manipulations of ndc_put and ndc_take buffers.
```
```
This example demonstrates that we can not only raise the level of abstraction of concurrency control, but we can relax and redefine its semantics to achieve improved concurrency. A typed local transaction log provides the necessary flexibility, acting as a medium for primitive operations to communicate with each other.

5 Discussion

Our vision, like that of other work on abstract concurrency control for transactional memory, is for concurrent software which reflects the structure of the application domain. Programming with locks and condition variables risks catastrophic structural failure as a system grows and requirements change. In contrast, composable memory transactions provide a safe foundation for evolution of concurrent software, but concurrency may suffer as transactions become more complex. Adding abstract concurrency control allows hot-spots to be removed with local changes only, and without changing the overall semantics of the program. Typed transactions provide a particularly flexible approach to abstract concurrency control.

Our system is still in the early stages of development, and much remains to be done. So far, our focus has been raising the level of abstraction for transactional programming, without sacrificing concurrency. This focus will continue as we experiment
with new data structures and concurrent programming patterns, but must eventually shift to other concerns. We are yet to examine performance. Our current implementation is high-level and na"ive, so we do not yet expect good absolute performance, nor to compete with other current STM implementations. Nevertheless, we believe that typed local transaction logs should provide the means to generalise many of the techniques used in high-performance STM implementations, to support abstract concurrency control. Some of the techniques we are interested in exploring include dependence-aware transactional memory (Ramadan et al. 2009) and aggressive transactional boosting (Koskinen & Herlihy 2008). We would also like to improve support for lock-based approaches (Dragojević et al. 2009), and contention management (Guerraoui et al. 2005). It remains to be seen whether it is possible (let alone desirable) to support multiple approaches to concurrency control in the one framework.

For many structures, like the finite map example (§4.2), development is likely to be similar to that of simple transactional variables. A natural evolution of the framework would be to factor these common parts into a library implemented in low-level C. Cooperation with the run-time thread scheduler may be beneficial for some parts, such as lock management.

Our development and presentation has so far been informal. Given the pitfalls which plague concurrent software, it will be particularly important to formalise our approach. This is not just to verify the soundness of the framework, but to provide correctness criteria to guide the development of new transactional structures. The formalisations given by Herlihy & Koskinen (2008) and Agrawal et al. (2008) give us confidence that this should be possible for a broad class of data structures. Our semantics would also need to account for modular blocking (retry) and choice (orElse) in the presence of open-nested transactions, and should accommodate structures with relaxed semantics, such as the non-deterministic channel (§4.3).

Certain features of the Haskell language are crucial to our design: its purity, and its ability to embed imperative sublanguages as abstract datatypes. Could our approach work in other languages? We think so, but probably only with language extensions. Thus, while our approach may only be accessible to Haskell programmers for the time being, we hope that in the longer term, it can contribute to the development of future concurrent programming languages. In particular, we believe that our work adds weight to arguments in favour of type systems which capture side-effects, particularly mutation, in a compositional way.

6 Conclusion

We have demonstrated our approach to abstract concurrency control, which allows concurrent data structures to be composed safely, without the loss of concurrency which affects transactional memories based on simple transactional variables. Our primary contribution is that transactions should be typed according to the structures they manipulate. Typed transactions admit abstract concurrency control without completely abandoning static safety guarantees. Typed local transaction logs also provide more flexible, and potentially more efficient abstract concurrency control.

References


