Tinkertoy® Transactions

Nicholas Haines Darrell Kindred J. Gregory Morrisett
Scott M. Nettles Jeannette M. Wing

December 1993
CMU-CS-93-202

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted to the Conference on Lisp and Functional Programmming 1994.

Abstract

We describe the design of a transaction facility for a language that supports higher-order functions. We factor transactions into four separable features: persistence, undoability, locking, and threads. Then, relying on function composition, we show how we can put them together again. Our "Tinkertoy" approach towards building transactions enables us to construct a model of concurrent, nested, multi-threaded transactions, as well as other non-traditional models where not all features of transactions are present. Key to our approach is the use of higher-order functions to make transactions first-class. Not only do we get clean composability of transactional features, but also we avoid the need to introduce special control and block-structured constructs as done in more traditional transactional systems. We implemented our design in Standard ML of New Jersey.

This research is sponsored in part by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330; and in part by National Science Foundation Fellowships for D. Kindred and J. G. Morrisett. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Wright Laboratory or the U. S. Government. The U. S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. This manuscript is submitted for publication with the understanding that the U. S. Government is authorized to reproduce and distribute reprints for Governmental purposes.

Keywords: transactions, threads, skeins, persistence, recovery, undoability, serializability, Standard ML, modules

1. Introduction

Transactions are a well-known and fundamental control abstraction that arose from the database community. They have three properties that distinguish them from normal sequential processes: (1) A transaction is a sequence of operations that is performed *atomically* ("all-or-nothing"). If it completes successfully, it *commits*; otherwise, it *aborts* and has no effects. (2) Concurrent transactions are *serializable* (appear to occur one-at-a-time), supporting the principle of isolation. (3) Effects of committed transactions are *persistent* (survive failures). Transactions can be nested.

The goal of our work is to provide modular support for transactions in a language that supports higher-order functions. By "modular" we mean

- *Factored.* Each key feature of transactions is supported independently of the others.
- *Composable.* Each individual feature can be composed with any other in a meaningful way. Furthermore, transactions themselves are composable with other features of the language.

As part of the Venari project, we chose to pursue these goals in the context of Standard ML of New Jersey [14]. SML/NJ supports higher-order functions, has a powerful modules facility, is freely available, and has an easily modified implementation. We broke transactions into these four separate features:

- *Persistence.* Effects of a computation can be made permanent, i.e., saved to disk.
- *Undoability.* Effects of a computation on the store can be undone.
- *Threads.* A computation may have multiple threads of control.
- *Locks.* Reader/writer (R/W) locks can be used to synchronize access to shared mutable data.

All but the last of these are useful as independent features and represent significant extensions to the semantics of SML. We package each feature into an SML module; each module exports some key higher-order functions. We then rely on higher-order function application to enable seamless composition of transactional features.

In the rest of this section we describe our modular approach to transactions and contrast it with a more traditional approach taken by the transaction community. In Section 2 we describe our design: the four building blocks in our model of transactions and how they compose. In Section 3, we explain how we express our design in SML. We close with discussions evaluating and summarizing our contributions. Throughout, we discuss related work in relevant sections.

Tinkertoy is a registered trademark of Playskool.

1.1. Our Approach

Essential to our approach is linguistic support for higher-order functions. Given a function `f` we want to be able to create a transactional version of `f` by applying the `transact` function to it. Thus,

```
(transact f) a
```

has the effect of applying `f` to `a` within a transaction. A more typical use is as follows:

```
((transact f) a)
    handle Abort => [some work]
```

The `Abort` exception handler allows some special action to be taken if the transaction aborts. Since `(transact f)` is simply a function and functions are first-class, our approach yields first-class transactions.

Most importantly, we want to be able to treat the function `f` as a black box. We want to be able to "wrap" `transact` around any `f` without changing the source code of `f` (or at most by applying a mechanical transformation to it). Someone else may have written `f`; it might even be multi-threaded. Without being able to simply wrap a transaction around a multi-threaded program, for instance, we would be forced to recode each separate thread in `f` as a concurrent nested transaction of a top-level transaction. This violates one aspect of modularity since the entire program would have to be recoded.

Consider a concrete (and the canonical) example where we want to transfer money from a savings account to a checking account in a bank. The transfer involves withdrawing money from the savings account and depositing it in the checking. We need to make sure that either both the withdrawal and the deposit succeed, or that neither of them occurs. So, we use a transaction to effect the desired behavior, where the actual work of the transfer is done by the `do_transfer` function:

```
fun transfer (savings, checking, amount) =
    let fun do_transfer () =
        (withdraw (savings, amount);     ***
         deposit (checking, amount))     ***
    in
        transact do_transfer ()
    end
```

We wrap a transaction around the `do_transfer` function so that if anything goes wrong, e.g., if `withdraw` raises an exception indicating that `savings` has insufficient funds, the whole transfer will be aborted. According to our semantics for `transact` (Section 3.2), if the transfer aborts, we re-raise the exception that caused the abort.

The `do_transfer` function could even be made multi-threaded by changing the starred lines above to the following:

```
(fork (fn () => withdraw (savings, amount));
 deposit (checking, amount))
```

Here the withdrawal and deposit are done in separate threads, but the transactional call to `do_transfer` stays the same.

1.2. The Traditional Approach

In contrast, a more traditional approach supported by transactional systems and languages such as CICS [10], R* [12], Camelot [6], Quicksilver [9], Argus [13], Arjuna [24], and Avalon/C++ [4], requires separate control constructs like `begin_transaction` and `end_transaction` to delimit a transaction's boundary.

For example, a skeleton of the bank transfer operation in Camelot would appear as follows [6]:

```
BEGIN_TRANSACTION
    ...
    if (savings_balance < amount) {
        ABORT(ERROR_INSUFFICIENT_FUNDS);
    }

    ... transfer money ...

END_TRANSACTION(status)

if (status == ERROR_INSUFFICIENT_FUNDS) {
    ...
}
```

There are several disadvantages to this approach. It requires syntactic extensions to the language to support transactions. Such textual extensions do not compose conveniently, nor can such transactions be manipulated as first-class values. Also the lack of exception handling forces the use of the special `status` variable. The programmer could easily forget to check the status after a transaction, in which case aborts would be ignored. Furthermore it is up to the programmer to propagate aborts in nested transactions.

2. Design Overview

Transactions may execute at the *top level* (Figure 1a), be *nested* inside one another (Figure 1b), or execute *concurrently* with each other (Figure 1c). Each may be *multi-threaded* (Figure 1d). The combination of all these kinds of transactions yields concurrent, nested, multi-threaded transactions (Figure 1e). In our pictures, we use a wavy line to denote a thread and a box to delimit the scope of a transaction; time advances from left to right. We appeal to tree terminology in discussing nested transactions: a transaction has a unique parent, a set of children, and sets of ancestors and descendants. A transaction is considered its own ancestor and descendant.

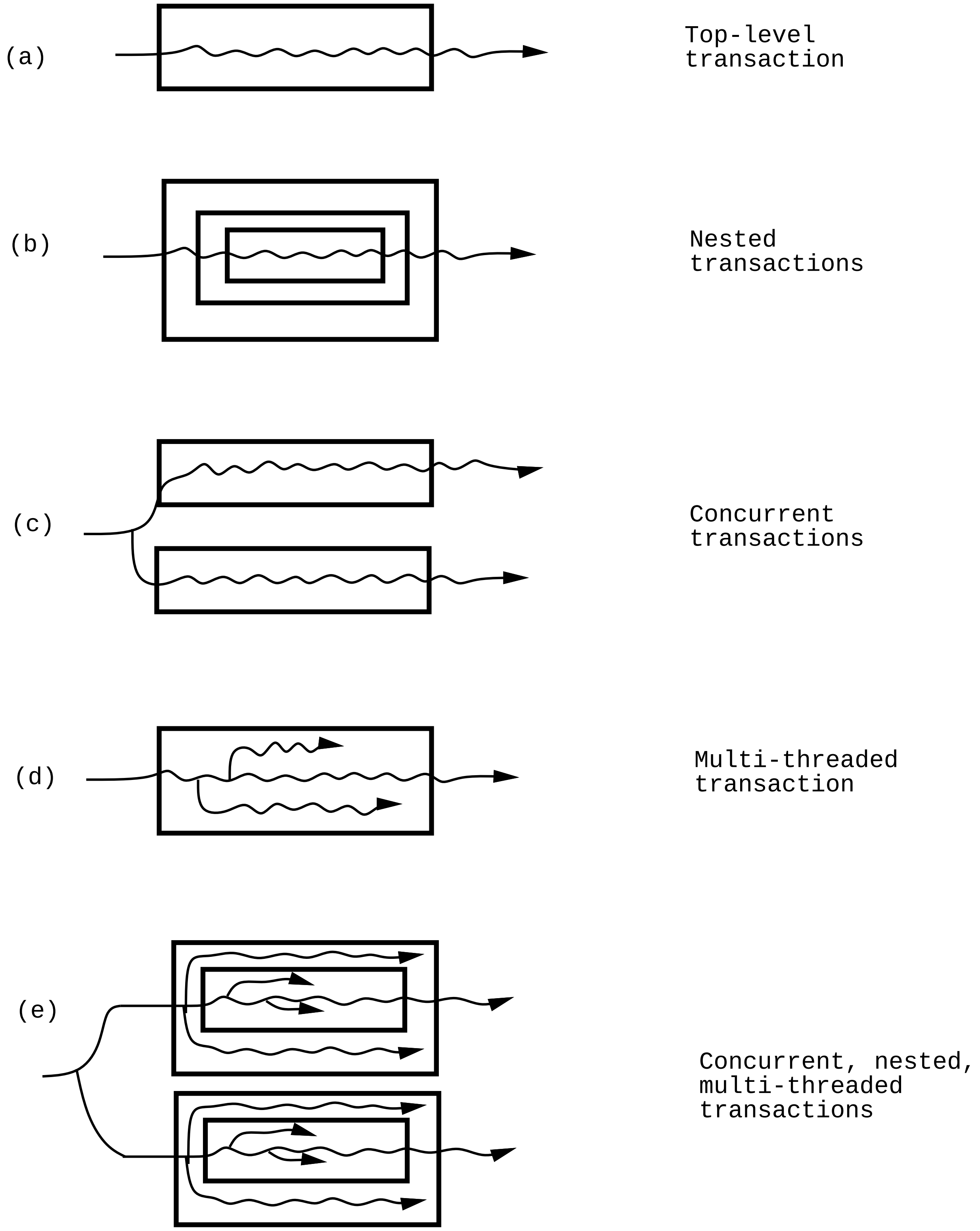

Figure 1: Nesting, Concurrency, and Multi-Threading

Since we separate the basic transactional features into individual components, we need to introduce terms that distinguish a regular transaction from one that supports some but not all features. A *regular transaction* is persistent, undoable, and locking. We use the term *persist-only transaction* for a computation that supports only persistence; we use the term *persistent transaction* for a computation that supports at least persistence. We use similar terms for *undo* and *locking*. When we say "transaction" unqualified, we mean a transaction of any kind (regular, persist-only, undo-only, locking-only, etc.). We will argue in Section 2.2 that all concurrent transactions need to be locking transactions as well.

In Section 2.1 we consider top-level and nested transactions of each flavor; in Section 2.2, we discuss concurrency, and more generally, different combinations of the features.

2.1. The Pieces

Persistence

A persistent value is one that outlives the computation that created it. We support a model of persistence popularized by the persistent programming language community [1]: *orthogonal persistence*. In this model, all data reachable by pointer dereferencing from a distinguished location, the persistent root, are persistent. Persist-only transactions cannot abort. Figure 2a depicts the execution of a function `f` in a top-level persist-only transaction; when it terminates, all persistent data modified by the transaction are saved to stable storage. If a crash occurs during the execution of `f`, we recover the last committed state from stable storage. All data not reachable from the persistent root are lost.

Only the effects of top-level persist-only transactions are made permanent; no action is taken when a nested persist-only transaction commits. We made this design decision in order to give a reasonable semantics to recovery from failure. If we were to recover partially-completed transactions, then we would have to be able to resume a transaction from the middle. This would be costly and difficult to achieve. Instead program failure or termination effectively aborts all transactions. No transactions need to be resumed from the middle upon recovery.

Undoability

A top-level undo-only transaction has no special effect if it commits. If it aborts then *all* changes it made to the store are undone. Our semantics for undo differs from traditional transactional systems in which only changes to persistent data (and not, for example, to any volatile data) are undone. Figure 2b depicts the execution of a function `f` whose effects may possibly be undone. At the start (conceptually) a checkpoint of the store is made. If it terminates successfully, then nothing unusual happens; if not, then `f`'s effects are rolled back to the checkpointed state, at which point a possibly different computation `g` can begin.

Undo-only transactions may commit or abort regardless of whether they are nested. However, since a nested transaction's commit is relative to the action of its parent, if the parent aborts then the effects of the (committed) nested transaction must be undone along with the parent's other

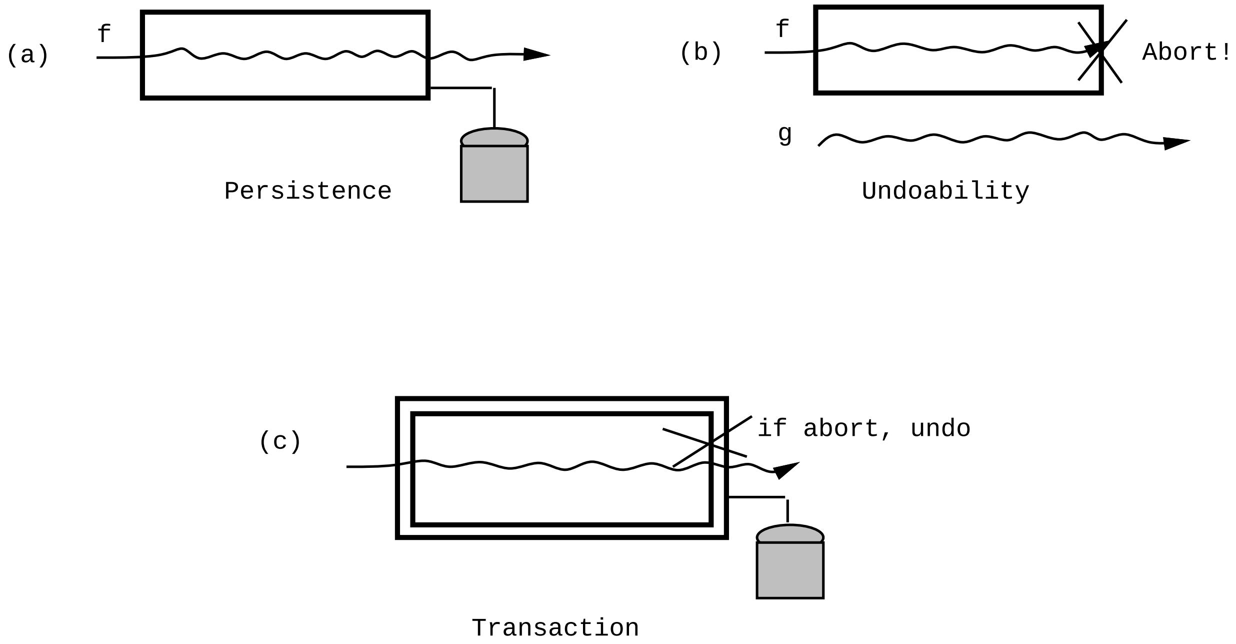

Figure 2: Persistence and Undoability

changes. Thus, when a child transaction commits it hands back ("anti-inherits") to its parent its set of changes to the store.

Threads

Threads are lightweight processes that communicate using shared mutable data and synchronize by acquiring and releasing mutual exclusion (mutex) locks. Individual threads may fork and start other computations, thereby providing a way to begin concurrent nested transactions.

We do not require threads within a transaction to be serializable; thus, they can engage in two-way communication using shared mutable data. Otherwise, we could not wrap `transact` around existing multi-threaded code without modification.

Locks

R/W locks are a well-known mechanism for ensuring serializability. Alone, they provide no support for commit or abort. Write locks also guarantee that any two concurrent transactions modify disjoint sets of data in the store, unless one is a descendant of the other.

In our model, when one thread of a transaction creates a child transaction, the other threads of the parent transaction are allowed to continue while the child executes. Many other transaction systems either force the parent to suspend while the child executes, or only allow transactions which have no children to modify data. We choose the less restrictive approach in order to keep the creation of transactions orthogonal to thread scheduling. It allows greater concurrency, as unrelated threads within a transaction do not get suspended whenever one of them creates a child transaction. In order to support this model, we use the following variation of Moss's standard locking rules for nested transactions [17]:

- A transaction may read a data object x if it holds a lock on x in read or write mode, and all writers are ancestors of the transaction.
- A transaction may write a data object x if it holds a lock on x in write mode, and all readers *and* writers are ancestors of the transaction.
- When a transaction commits, all its locks are anti-inherited, i.e., handed off to its parent. They are released if the transaction is top-level. If the transaction aborts, all its locks are released.

2.2. Putting the Pieces Together

Nesting enables us to construct a top-level regular transaction from an undo-only transaction nested inside a top-level persist-only transaction (Figure 2c). If the undo-only transaction commits then all changes to the stable store are saved by the persist-only transaction. If the undo-only transaction aborts, all changes are rolled back. Thus when the persist-only transaction saves all changes to the stable store there will be no changes on behalf of the aborted undo-only transaction to save; the net effect is that the stable store is in the same state as at the beginning of the transaction.

More generally, each combination of the different kinds of transactions has a well-defined meaning. For example, an undo-only transaction can have a persist-only transaction nested within it, and vice versa. A transaction can have nested within it concurrent transactions of different flavors.

To support complete "mixing-and-matching" of features, however, we need to impose two rules, one to deal with concurrency and one to deal with arbitrary nesting:

1. All accesses to data shared among concurrent transactions (of any flavor) must be coordinated by R/W locks.
2. We delay writes to stable storage until the commits of top-level transactions.

Let's argue the case for the first rule by considering a top-level undo-only transaction S that executes concurrently with some other top-level transaction T. If S and T modify the same object x they may interfere with each other. To ensure that they do not, each transaction must acquire a R/W lock for x and hold onto the lock until the transaction completes.

To see why, suppose S and T synchronize using only a mutex lock such that S acquires a mutex lock for x, modifies x to be the value x_S, and then immediately releases the mutex lock. S then continues to execute while T acquires the mutex lock for x, modifies x to be the value x_{ST}, and releases the lock. Now suppose S aborts and T commits. What should the value of x be? Certainly it cannot be x_S since S's effects must be undone. It cannot be the original value of x because then T's effects would be undone. It cannot be x_{ST} since T's modification to x may have depended on the value that it observed after S changed x. Ideally, x's value should be the result of executing T

again on the original value of x. This means that the effects of both S and T have to be undone and then T has to be reexecuted; clearly, this solution is impractical (considering that there may be many more concurrent transactions than just T, each of which may have depended on T's commit). A more reasonable solution would be to require that S not release the mutex lock until after it has completed, either aborting or committing. Since the properties of R/W locks provide exactly the semantics needed, we require that all concurrently running transactions synchronize using R/W locks. (Threads within a transaction still need only synchronize using mutex locks since we do not require that they be serializable.)

Since a similar argument holds for concurrent, top-level, persistent transactions, and since top-level, locking transactions are locking by definition, all concurrent, top-level transactions must coordinate using R/W locks. Use of R/W locks also ensures that concurrent nested transactions are isolated from one another.

Let's now argue the case for the second rule. Consider the case of a persist-only transaction P nested inside an undo-only transaction U. If U commits then all its changes to the store are kept and the state of stable storage should reflect any changes up to the point when P committed. If U aborts then all changes to the store should be undone and the state of stable storage should be unchanged. However, if we had allowed P to write its changes to the stable store when it committed, we would then have to roll back the stable store. Fortunately, we can achieve the desired semantics by delaying P's commit to stable storage until after U commits or aborts. If U commits, then all changes (including P's) are saved to stable storage; if it aborts, then all changes (including P's) are undone and the stable store is not changed. In general, a persistent transaction's effects are not written to stable storage until its top-level ancestor commits. Primarily for uniformity, in the case of any persistent transaction P nested inside a locking-only transaction, we choose also to delay the commit of P until the commit of P's top-level ancestor.

Finally, consider the behavior of threads that are outside of any transaction. Programmers using such threads should not expect strong consistency guarantees; otherwise they should use transactions. Such threads have no interaction with undo; their effects cannot be undone. Such threads may modify the persistent store but since they do so outside of a persistent transaction programmers cannot expect these changes to be immediately reflected in stable storage. We choose to write such changes to stable storage whenever a persistent transaction completes; we must do such writes at these times because the committing transaction may have depended on the value of persistent data modified by the thread. Finally, threads outside of transactions that wish to communicate with transactions can do so only through mutex locks, and therefore should be used with care or avoided altogether since use of such locks does not guarantee serializability; other transactional facilities that allow threads to exist outside transactions, e.g., Camelot [6] and Encina [5], have similar caveats.

3. Expressing our Design in SML

We are able to express our design in a very simple, straightforward, and elegant manner in SML. In the next three sections we first individually describe the SML interfaces for the four transactional building blocks, then show how we put them all together, and then show how we can use our constructs to implement the bank example. Implementation details for persistence are discussed in greater detail by Nettles and Wing [19]; for undoability, by Nettles and Wing [19] and Morrisett [16]; and for threads in SML, by Cooper and Morrisett [3]. In Figures 3–6 we show only the portions of the PERS, UNDO, RW_LOCK, and THREADS interfaces that are relevant to this paper. The Venari/ML technical report gives further details of these interfaces and examples showing their use [27].

3.1. The Pieces

Persistence

```
signature PERS = sig
    val persist  : ('a -> '_b) -> 'a -> '_b

    val bind     : identifier * 'a -> unit
    val unbind   : identifier -> unit
    val retrieve : identifier -> 'a
    ...
end
```

Figure 3: PERS Interface

The key higher-order function exported by PERS is persist.[1] The expression (persist f) a has the effect of evaluating f a. If it is the outermost call of persist and f a terminates, f's changes to persistent data are saved to disk. If f does not terminate, e.g., a crash occurs during its execution, none of f's changes are saved.

All data reachable from the persistent root are persistent, and thus, recoverable. Any first-class SML value can be made persistent simply by arranging that it be reachable from the persistent root. The other functions of PERS allow manipulation of a symbol table that stores bindings between identifiers and values; the table itself is reachable. Thus, we can store and retrieve persistent values by name.

Our implementation uses a separate persistent heap to store all values reachable from the persistent root. Modifications to these values may cause values in the volatile heap to become reachable as well. On commit, any newly reachable values must be moved into the persistent heap, and all modifications to persistent values must be written to stable storage. We use the Recoverable Virtual Memory system [23] to provide an efficient implementation of stable storage based on logging.

Undoability

UNDO exports the undoably function, which allows users to make undoable changes to the store, an essential feature of a transaction that may abort. The undoably function is a wrapper for any function f such that if the exception Restore is raised while executing f, all of f's effects on the store are undone; undoably f behaves exactly like f if no exception is raised. The changes undone include those done within any nested transactions.

[1] We use the underscore character, as in '_a and '_b, for weak (imperative) type variables [14].

```
signature UNDO = sig
    val undoably : ('a -> '_b) -> 'a -> '_b
    exception Restore of exn
    ...
end
```

Figure 4: UNDO Interface

The semantics of `undoably` is defined only with respect to the store. In particular, a transaction's effects through I/O (e.g., writing to a terminal) are not undone.

We implement undo by logging the location and old value of every mutation. Upon abort we replay the log and restore the old values. To anti-inherit changes to the store we splice the child transaction's log onto the parent's log.

In most imperative languages this implementation would have unacceptable performance. In SML/NJ it works well for several reasons. First, assignments are relative rare. Second, the locations of many assignments are already logged to support generational garbage collection. We have simply extended these logs to capture all assignments and to record old values.

Our implementation for both persistence and undoability assumes that concurrent transactions modify disjoint sets of data in the store; this assumption is easily discharged by our first rule (Section 2.2) that concurrent transactions use write locks for accessing data.

R/W Locks

A. Locks

```
signature RW_LOCK = sig
    eqtype rw_lock

    exception Read
    exception Write

    val rw_lock       : unit -> rw_lock
    val acquire_read  : rw_lock -> unit
    val acquire_write : rw_lock -> unit
    val read          : rw_lock -> ('a -> 'b) -> 'a -> 'b
    val write         : rw_lock -> ('a -> 'b) -> 'a -> 'b
    ...
end
```

Figure 5: RW_LOCK Interface

We provide R/W locks to enable the programmer to enforce isolation and serializability among concurrent transactions. These locks are held per transaction. On commit, they are handed off to the parent transaction (if any); on abort, they are simply released.

A lock is created by a call to `rw_lock`. It is acquired for reading or writing by a call to `acquire_read` or `acquire_write` respectively. A thread within a transaction can perform reads and writes on the data protected by a lock, subject to our variation of Moss's rules stated in Section 2.1.

The `read` and `write` functions take a lock, a function, and its argument, and apply the function to the argument with the guarantee that the conditions specified by the rules will hold during the execution of the function. Since we allow transactions to execute concurrently with their children, the implementation must ensure that no descendant of the calling transaction may acquire the lock while the operation is in progress. This is currently achieved by prohibiting all other threads from performing any operation on the lock (including `read` and `write`) while the function executes. This is not a serious problem since the function is expected to be a simple, constant-time operation such as assignment. A more sophisticated implementation could allow somewhat greater concurrency, but the cost of the added complexity would likely outweigh the gain in concurrency. If a transaction calls `read` or `write` without holding the lock in the appropriate mode, the exception `Read` or `Write` will be raised; otherwise, `read` and `write` will block until the condition is satisfied.

B. Safe State

```
signature RW_REF = sig
    type 'a rw_ref
    type rw_lock

    val rw_ref  : '_a * rw_lock -> '_a rw_ref
    val rw_get  : 'a rw_ref -> 'a
    val rw_set  : 'a rw_ref -> 'a -> unit
    val lock_of : 'a rw_ref -> rw_lock
    ...
end
```

The only mutable data types in SML are refs and arrays. Thus, it is easy for us to provide two structures to help the user manipulate state safely [25, 15]. *Reader-writer refs* (`RW_REF`) are refs protected by R/W locks; in order for a transaction to access these objects, it must hold the `rw_lock` (for reading or writing, as appropriate).

The above `RW_REF` signature parallels the SML pervasive `REF` signature. The accessing functions (`rw_get`, `rw_set`) call `RW_Lock.read` or `RW_Lock.write` to ensure that the read or write condition holds. If the lock is not held in the appropriate mode, the `RW_Lock.Read` or `RW_Lock.Write` exception is raised. The `lock_of` function returns the lock associated with a `rw_ref`.

Arrays are handled completely analogously.

```
signature THREADS = sig
    val fork    : (unit -> unit) -> unit

    type mutex
    val mutex   : unit -> mutex
    val acquire : mutex -> unit
    val release : mutex -> unit
    ...
end
```

Figure 6: THREADS Interface

Threads and Skeins

A. Threads

The THREADS module exports essential functions for creating a thread, and for acquiring and releasing mutex locks. Other functions, not relevant here, support manipulating condition variables and thread state. Our interface is similar to other Threads packages for C [2], Modula-2+ [22], and Modula-3 [8].

The function `mutex` creates a new mutex value. The function `acquire` attempts to lock a mutex and blocks the calling thread until it succeeds. At most one thread may hold a given mutex at any time. The function `release` unlocks a mutex, giving other threads a chance to acquire it. Unlike R/W locks, mutex locks are short-term, i.e., they are not held for the duration of a transaction. Programmers have complete control over when to release them.

B. Skeins

We introduce a new abstraction, called a *skein*, for controlling threads as a group. Conceptually a skein implements each of the boxes drawn in Figures 1 and 2. Within a skein some SML function (the *body* of the skein) is executed. The body itself may fork threads. We assume a barrier synchronization model for skein termination. The skein will not finish until the body thread returns a value and all other threads have finished; only one thread ever leaves a skein. All held mutexes must be released before return. If any thread (including the body thread) running inside a skein raises an uncaught exception, the skein aborts. Any extant forked threads and child skeins are killed, and the exception is propagated to the outside. A skein also holds R/W locks that are shared among its threads.

A transaction needs to execute certain code within a skein (while the R/W locks are still held), but *after* all threads within that skein have completed or died. Such code might, for example, commit persistent changes to disk or release R/W locks. Thus, our skein abstraction has the following interface:

```
signature SKEINS = sig
    datatype 'a result = Result of 'a
                       | Exception of exn
    exception Abort

    val skein:
        (unit -> unit) ->                (* initializer *)
        ('_b result -> '_b result) ->    (* completer *)
        ('a -> '_b) ->                   (* body *)
         'a -> '_b                       (* result *)
end
```

The body of a skein is executed in a sub-thread within the skein, while a *control thread* waits for it to complete. The first two arguments to `skein` are (1) an initializer function, which is called in the control thread before the body thread is forked; and (2) a completer function, which is called in the control thread after the body has returned and any extant threads have ended. The completer is applied to the value returned by the body or the exception that caused premature termination, and it returns a `result` value that is in turn presented as the result of the call to `skein`.

If the body of a skein finishes while sub-skeins are still executing, the sub-skeins are terminated, calling their completer functions with the `Abort` exception. The parent skein's completing function is not called until all sub-skeins have completed.

We use skeins to implement multi-threaded transactions of all kinds, e.g., persist-only and undo-only transactions, by passing in appropriate initializer and completer functions.

```
signature VENARI = sig
    val transact : ('a -> '_b) -> 'a -> '_b

    structure Pers     : PERS
    structure Undo     : UNDO
    structure RW_Lock  : RW_LOCK
    structure RW_Ref   : RW_REF
    structure RW_Array : RW_ARRAY
    structure Skeins   : SKEINS
    structure Threads  : THREADS
end
```

Figure 7: Transaction Interface

3.2. Putting It All Together

Putting all these pieces together into a single SML module culminates in our main `VENARI` interface, shown in Figure 7. It provides a way for application programmers to create and manipulate concurrent, nested, multi-threaded transactions. A transaction is a *locking skein* of *threads* whose effects are *undone* if the transaction aborts or made *persistent* if it terminates normally. We require that each transaction access only safe state.

The various features described in the previous sections are all used in VENARI's main function, transact, which evaluates its argument within a transaction. We implement transact as a special case of skeins, reusing the initializer and completer functions defined for persist-only and undo-only transactions:

```
val init_transact = Undo.init_undo o Pers.init_pers

val complete_transact = Pers.complete_pers o Undo.complete_undo

val transact = Skeins.skein init_transact complete_transact
```

3.3. Implementation of the Bank Example

We give an implementation of a bank account in Figure 8. L is the Venari.RW_Lock substructure, R is Venari.RW_Ref, and V is Venari.

The account is a ref to a real (initially 0.0), protected by a R/W lock. Assuming that amount is non-negative, the deposit function first acquires the lock associated with the account in write mode; it then updates the account's value to the sum of the old value and the new amount. The withdraw function is slightly more complicated since it needs to check whether there is sufficient money in the account before the withdrawal occurs. Raising the unhandled Insufficient_Funds exception would cause the transaction to abort.

Using this interface we can do a bank transfer as described in Section 1.1.

4. Evaluation

In the introduction we stated two goals of our work: factoring transactions into individual features and composing these features with each other and with other features of SML. For the most part, we succeeded in accomplishing both goals and are able to express our results very concretely through our Venari/ML interfaces. In this section we evaluate the successes and limitations of our work.

We achieve composability by making transactions with higher-order functions. Making transact higher-order means that transact can easily be used as a wrapper function. This kind of composability facilitates code reuse. For example, suppose we have an interface along with a non-transactional implementation. We can implement a transactional version of this interface by wrapping a transact around each of the non-transactional functions without any knowledge of their internal structure.

The one feature of the New Jersey implementation of SML with which our transactional extensions do not interact well is call/cc. We use it extensively in our implementation of threads. However, we cannot export it to the user because it does not interact well with SML's exception handling, which we use to deal with aborted transactions in a graceful manner. Unfortunately, when a continuation is invoked, a new exception handler context is installed. Consequently, we cannot guarantee that a computation will pass through our handlers. For example, if call/cc were exported to the user, we could not guarantee that a skein's completer function would be called. An alternative solution would be to implement a mechanism similar to Scheme's unwind-protect [7, 21].

```
functor Account (structure Venari: VENARI)
      : ACCOUNT = struct

    type account = real R.rw_ref

    fun new_account () =
        R.rw_ref (0.0, L.rw_lock())

    fun deposit account amount =
        let fun do_deposit () =
            (L.acquire_write (R.lock_of account);
             R.rw_set account ((R.rw_get account) + amount))
        in
            V.transact do_deposit ()
        end

    exception Insufficient_Funds

    fun withdraw account amount =
        let fun do_withdraw () =
            (L.acquire_write (R.lock_of account);
             let val bal = (R.rw_get account)
             in
                 if bal < amount then
                     raise Insufficient_Funds
                 else
                     R.rw_set account (bal - amount)
             end)
        in
            V.transact do_withdraw ()
        end

end
```

Figure 8: Bank Account Implementation

We also successfully achieved a factorization of transactions into their component parts. We found that to allow transactions of any type to execute concurrently requires the use of R/W locks. That support for concurrency needs support for synchronization should come as little surprise. More surprising was that we were successful at decoupling the other three features from each other.

One advantage of decoupling transaction features is that each feature can be used independently. For example, undoability is useful for implementing backtracking search. Typical Prolog implementations use an explicit mutation log, called a *trail*, which is used to undo variable bindings when backtracking [26]. Undo would allow the elimination of the trail and allow the desired functionality to be expressed directly using `undoably`.

Another benefit of this factorization is that it helped us recognize new abstractions. In our implementation of undo, the ability to save and restore the state of the store is implicit and governed by rules about nesting transactions. Inspired by this work and the semantics of imperative programming languages, Morrisett has recently proposed and implemented a new programming language feature, *first-class stores* [16]. In his system the current store can be captured and saved away like any other first-class value. At any later point during the program's execution the saved store can be restored.

A final benefit to factoring our design has been in factoring our implementation. Our original implementation of the persistence and undo subsystems was factored largely for convenience of implementation. At the same time, our original threads implementation was built completely independently of the transaction system. Surprisingly, adding support for concurrent, multi-threaded transactions has not forced these implementations to merge and become monolithic. Instead the mutation log serves as a common data structure used independently by the undo and persistence subsystems, and is maintained on a per transaction basis. Needless to say the factored nature of the implementation has made it easier to build and maintain.

We have not yet attempted to design and implement support for distributed transactions. If we were to attempt such support, our factored implementation as well as the notion of first-class stores mentioned above would come in handy. Committing a distributed transaction requires a two-phase protocol. In the first phase the current state of the transaction must be made persistent in such a way that it can be undone. We can achieve this effect by capturing the store as a first-class value and then making that value persistent. Given support for any kind of distribution, adding distributed transactions should be straightforward.

Two other limitations of our work result from deliberate decisions (1) not to explore support for other ways of ensuring serializability besides R/W locks, e.g., using timestamps, and (2) not to attempt to give semantics to undo for I/O (as in undoing the dispensing of cash from an ATM machine). These issues have been thoroughly addressed by the database community.

Finally, we have made significant progress in measuring and improving the performance of our system. Recently O'Toole, Nettles, and Gifford added a concurrent garbage collector for the persistent heap [20]. They show that the performance of both the collector and the persistence subsystem is good—comparable to a simpler system that supports neither orthogonal persistence nor garbage collection. Nettles is currently completing a more thorough performance evaluation that will allow us to improve the performance of our system substantially [18].

5. Summary of Contributions

The main contribution of our work is to show that transactions can be broken into separable components, each supporting a different aspect of a traditional transactional model: persistence, undoability, locking, and threads. These components can then be composed to build the traditional model or even new models with weaker semantics.

Two technical ideas resulted from pushing hard to achieve our goal of composability. One is the idea of a new control abstraction, skeins, with which we can build variations of the transactional model as simple special cases. The other is a set of guarantees, captured by our variation of Moss's rules, that gives a reasonable semantics to nested, multi-threaded transactions. Heretofore other systems either permit only a single thread of control to execute with a transaction [13, 4] or support multi-threaded transactions with no semantic guarantees [6, 5]. Except for Humm [11] we are not aware of any other work that attempts to give nested, multi-threaded transactions a semantic basis.

A more concrete contribution is our specific set of extensions to SML/NJ in support of concurrent, nested, multi-threaded transactions. In our design, we exploited SML's higher-order functions and modules facility. We use its exception handling mechanism to give control to the programmer in case a transaction aborts. Our implementation uses the New Jersey implementation of SML in some critical ways, e.g., its support for `call/cc` and the logs used by its garbage collector. Our current implementation is based on SML/NJ (0.80) and runs in the Mach 2.5 environment.

By adding such extensions to an advanced programming language like SML, we have provided application programmers with some high-level constructs (above the operating-system level) to use transactions unintrusively. By using simple wrapper functions, programmers need not worry about formatting and unformatting data into files in order to achieve persistence; they can undo effects to the store if desired (e.g., for backtracking); and they have explicit control over concurrent access to shared mutable data through mutex and R/W locks.

Acknowledgments

We thank Manuel Faehndrich for his participation in the initial design and implementation of concurrent, multi-threaded transactions, and Karen Kietzke for her help in maintaining our current implementation.

References

[1] M.P. Atkinson, P.J. Bailey, K.J. Chisolm, W.P. Cockshott, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4):360–365, 1983.

[2] E.C. Cooper and R. Draves. C Threads. Technical Report CMU-CS-88-154, Carnegie Mellon Computer Science Dept., 1988.

[3] E.C. Cooper and J. Gregory Morrisett. Adding threads to Standard ML. Technical Report CMU-CS-90-186, Carnegie Mellon School of Computer Science, December 1990.

[4] D. L. Detlefs, M. P. Herlihy, and J. M. Wing. Inheritance of synchronization and recovery properties in Avalon/C++. *IEEE Computer*, pages 57–69, December 1988.

[5] Graeme Dixon. Encina (Transarc corporation). Private communication, 1993.

[6] J. Eppinger, L. Mummert, and A. Spector. *Camelot and Avalon: A Distributed Transaction Facility.* Morgan Kaufmann Publishers, Inc., 1991.

[7] Friedman and Haynes. Constraining control. In *Proceedings of ACM Symp. on Prog. Lang.*, 1985.

[8] S.P. Harbison. *Modula-3.* Prentice-Hall, 1992.

[9] R. Haskin, Y. Malachi, W. Sawdon, and G. Chan. Recovery management in QuickSilver. *ACM Transactions on Computer Systems*, 6(1):82–108, February 1988.

[10] P. Helland. Transaction monitoring facility. *Database Engineering*, 8(1):9–18, June 1985.

[11] Bernhard G. Humm. An extended scheduling mechanism for nested transactions. In *Int'l IEEE Workshop on Object-Orientation in Operating Systems*, Ashville, NC, December 1993.

[12] B.G. Lindsay, L.M. Haas, C. Mohan, P.F. Wilms, and R.A. Yost. Computation and communication in R*: A distributed database manager. *ACM Trans. on Comp. Systems*, 2(1):24–38, February 1984.

[13] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Language and Systems*, 5(3):382–404, July 1983.

[14] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[15] J. Gregory Morrisett and Andrew Tolmach. Procs and locks: A portable multiprocessing platform for Standard ML of New Jersey. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego*, pages 198–207, May 1993.

[16] J.G. Morrisett. Generalizing first-class stores. In *Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages*, pages 73–87, June 1993.

[17] J.E.B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* The MIT Press, 1985.

[18] Scott M. Nettles. Safe and efficient transaction support for heap-based languages. Ph.D thesis in progress.

[19] Scott M. Nettles and J.M. Wing. Persistence + Undoability = Transactions. In *Proc. of HICSS-25*, January 1992. Also CMU-CS-91-173, August 1991.

[20] J.W. O'Toole, S.M. Nettles, and D.K.Gifford. Concurrent compacting garbage collection of a persistent heap. In *Proceedings of the 14th ACM Symp. on Operating System Principles*, December 1993.

[21] Jonathan Rees. The Scheme of things: The June 1992 meeting. *LISP Pointers*, 5(4):40–45, 1992.

[22] P. Rovner. Extending Modula-2 to build large, integrated systems. *IEEE Software*, 3(6):46–57, 1986.

[23] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight Recoverable Virtual Memory. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, December 1993.

[24] S.K. Shrivastava, G.N. Dixon, F. Hedayati, G.D. Parrington, and S.M. Wheater. A technical overview of Arjuna: A system for reliable distributed computing. Technical report, Computing Laboratory, University of Newcastle upon Tyne, 1988.

[25] Andrew P. Tolmach and Andrew W. Appel. Debuggable concurrency extensions for Standard ML. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 120–131, May 1991. Published as *SIGPLAN Notices* 26(12), December 1991. Also Princeton Univ. Dept. of Computer Science Tech. Rep. CS-TR-352-91.

[26] D.H.D. Warren. An abstract PROLOG instruction set. Technical Report 309, AI Center, SRI International, Menlo Park, 1983.

[27] J.M. Wing, M. Faehndrich, N. Haines, K. Kietzke, D. Kindred, J.G. Morrisett, and S. Nettles. Venari/ML interfaces and examples. Technical Report CMU-CS-93-123, Carnegie Mellon Computer Science Department, March 1993.