Software Transactions Meet First-Class Continuations

Aaron Kimball   Dan Grossman
University of Washington
{ak,djg}@cs.washington.edu

Abstract

Software transactions are a promising technology that make writing correct and efficient shared-memory multithreaded programs easier, but adding transactions to programming languages requires defining and implementing how they interact with existing language features. In this work, we consider how transactions interact with first-class continuations. We demonstrate that different idiomatic uses of continuations require different transactional semantics, so a language supporting transactions and `call-with-current-continuation` should provide programmers with a way to control these semantics. We present a design meeting this need, addressing both escaping from and reentering the dynamic extent of a transaction.

We have implemented our design by modifying Scheme48. We present the most interesting details of the implementation and its performance on some small benchmarks.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Concurrent programming structures

General Terms Design, Languages

Keywords Atomicity, Transactions, Continuations, Scheme

1. Introduction

Software transactions provide programmers of shared-memory multithreaded systems with a straightforward synchronization mechanism that is easier to use than locks and condition variables. The key idea behind a programming-language transaction is that it completes a computation *as though* there are no interleaved threads while the underlying implementation still ensures fair scheduling. For example, using *atomic* as a new primitive that takes a thunk and evaluates it as a transaction, the following procedure swaps the contents of a `cons` cell without another thread being able to observe any intermediate state:

```
(define (thread-safe-swap! pr)
  (atomic (lambda ()
     (let ((x (car pr)))
        (set-car! pr (cdr pr))
        (set-cdr! pr x)))))
```

Proceedings of the 2007 Workshop on Scheme and Functional Programming
Université Laval Technical Report DIUL-RT-0701

This code is correct without appealing to any locking protocol or placing restrictions on the rest of the program.

More interesting examples may use diverse features from the underlying language. It is important to allow as many features as possible during evaluation of an atomic expression so that we can reuse libraries, maintain procedural abstraction, etc. With help from the language implementation, mutation, function calls, and memory allocation can all be supported in transactions.[1] However, for transactions to fulfill their promise as a next-generation synchronization mechanism, there must exist a well-defined interaction between transaction primitives and a language's control operators.

Previous work on transactions in functional and object-oriented languages has focused on one question in this space: What does it mean if an exception thrown from within a transaction is caught outside a transaction? This paper investigates a more general question: *What should it mean if the invocation of a first-class continuation crosses the boundary of a transaction's dynamic extent?*

Rather than provide a single definitive answer, we conclude that the best semantics depends on how the continuation is being used. For example, the natural behavior for a coroutine call is not the best choice for an exceptional escape and vice-versa. Therefore, we have designed transactional support for Scheme that allows programmers to specify the behavior they wish. To do so, we have provided a prototype that permits many reasonable behaviors so that we and the community can experiment with the interplay between transactions and continuations. Though programmers must specify a behavior (or inherit predefined defaults), this is typically done when entering a transaction or creating a continuation. In this way, we preserve the Scheme design decision that invocations of continuations appear as ordinary function applications.

To validate the feasibility and convenience of our design, we modified Scheme48 [18] to support software transactions. As a pleasant by-product independent of continuations, we believe this work provides the first full-fledged implementation of transactions in Scheme, albeit one supporting only a uniprocessor rather than true parallelism. While the basic approach is much like the second author's previous work on extending Objective Caml [24], the support for continuations is entirely new and modifying an R^{5}RS [1] interpreter has its own unique challenges. As we discuss, all the examples in this paper run correctly and efficiently on our prototype, which is publicly available.[2]

In short, our contributions are:

- A taxonomy of the ways first-class continuations can interact with transactions and the programming idioms that lead to the different interactions
- A language design that gives programmers control over the interaction between continuations and transactions

[1] External actions (I/O) can generally not be, as discussed in Section 2, and this is not our present focus.

[2] http://www.cs.washington.edu/homes/ak/atomscheme/

- An extension to an R^{5}RS implementation supporting software transactions
- A preliminary evaluation showing our design is efficiently implementable

The rest of this paper proceeds as follows. Section 2 discusses related work, focusing on the interaction between transactions and exceptions as well as recent work on concurrency in Scheme. Section 3 provides context by discussing the high-level aspects of the design and implementation of our transaction system that are not directly related to continuations. Section 4 describes different idiomatic uses of continuations and how they best interact with transactions. Section 5 presents our language design encompassing these possibilities and describes the primitives' semantics. Section 6 presents the most interesting pieces of our Scheme48-based implementation, including its current status and performance characteristics. Section 7 discusses future work and concludes.

2. Related Work

Given the vast amount of research on transactional memory and language support for concurrency in recent years (see, e.g., the recent overview by Larus and Rajwar [19]), we focus on only the most relevant work. This includes full language designs and implementations (Section 2.1), work considering the interaction with exceptions (Section 2.2), and recent work on concurrency in Scheme (Section 2.3). The most notable omissions are hardware transactional memory and library-based approaches. The latter requires no changes to the compiler but programmers must use library functions to access memory within transactions.

2.1 Language Implementations with Transactional Memory

Harris and Fraser [12] were the first to provide an *atomic* primitive for a modern language (Java) and an implementation that scaled on multiprocessors. Unlike older work, this primitive does not require programmers to acquire locks [10] nor identify what data might be accessed within a transaction [20]. Implementation-wise, Harris and Fraser maintain a private version of accessed memory for a transaction and reflect updates back to shared memory via a highly nontrivial commit protocol that facilitates parallel commits. Later work [14, 2, 26] improves performance for Java and C# by (1) performing updates on main memory directly, using exclusive ownership to prevent race conditions and undoing the updates if the transaction conflicts with a parallel operation, and (2) using compiler optimizations or novel hardware [5] to reduce the overhead of conflict detection.

Work on adding transactions to Objective Caml [24] and to Real-Time Java [21] established that the overhead for transactional memory is negligible if one assumes that at most one thread runs at a time, an assumption that holds on uniprocessors and is already made by many language implementations. As reviewed in Section 3, this assumption lets one incur no overhead for any operation except an imperative update inside a transaction.

Transactional memory has also been added to Haskell [13], where a monad cleanly encapsulates mutable memory accessible within transactions and an *orelse* combinator lets programmers try an alternative transaction if another one aborts. Transactions are also included in several next-generation high-performance computing languages [3, 6, 7], though implementations are still in flux.

2.2 Transactions and Exceptions

Several projects have considered the question of what should happen if an exception causes control to transfer from within a transaction to a handler outside the transaction. In prior work [24, 15], such exceptions are considered to commit the transaction, after which control is transferred to the correct exception handler. This is consistent with Harris and Fraser's original work [12] and implements the semantics of exceptions as non-local jumps carrying values.

Others have argued that having such exceptions abort the transaction avoids the need for cumbersome code that compensates for an error-condition. Instead, the memory-rollback inherent to transactions naturally reverts state to the pre-transaction version. The work in Haskell [13] therefore interprets an exception within a transaction as a *retry*, i.e., abort the transaction and transfer control back to its beginning.

More complicated variants undo memory updates but still propagate control to the outer exception handler. As Section 4 explains in more detail, this induces awkward semantics because the actual exception value might reference memory locations whose contents were rolled back. The Fortress language [3] does not roll back memory allocated since the transaction was entered, noting that this decision affects only the exception value. Fortress also has a variant of *atomic* in which the user can undo the memory updates and abort to the control point just after the transaction. Slightly different is Harris et al.'s approach of making a deep copy (via Java/C# serialization) of the exception value before undoing memory updates [11, 25]. They also note that such semantics is useful even in single-threaded code. We have implemented yet a third subtle variation for Scheme in which imperative updates are reverted so objects allocated in the transaction will have their initial values.

To our knowledge, no prior work has considered a control transfer into a transaction's dynamic scope, which cannot occur with exceptions. Our design for Scheme encompasses most of the above variations while also supporting such resumptive continuations.

2.3 Concurrency in Scheme

The most closely related work in Scheme is the *proposals* structure for optimistic concurrency in Scheme48 [17]. Transaction-specific primitives are used to read and write `cons` cells, vectors, etc., and a log is used to record all such provisional reads and writes in private memory until attempting to commit. In contrast, our approach writes to shared memory eagerly and undoes the update if the transaction fails or is preempted. The proposals work has several limitations we avoid. First, that work requires separate primitives for memory access within transactions, which prevents reusing libraries inside and outside transactions. Using a nonprovisional primitive inside a transaction could lead to subtle errors. Second, the proposals library has no provisional `set!`. Third, the commit protocol is explicitly subject to "A-B-A" concurrency errors: it checks only that locations read during the transaction have the same value at the commit-point, but it is well-known that this is insufficient to prevent data races. Fourth, there is no support for interaction with continuations; it is up to the programmer to determine which continuation-invocations cross a transaction boundary and use low-level primitives to change relevant state as to whether a transaction is running, should be committed, etc.

Gasbichler and Sperber's work on integrating threads with UNIX-style processes in scsh [9] is related in that it considers the interaction between mutable bindings and continuations. However, the semantics of UNIX fork is exactly the opposite of shared-memory: resources are copied. Difficult semantic questions arise, such as what should happen to a continuation's bindings when it is invoked in a different process than its creator. The solutions do not appear directly relevant to the transactional-memory setting.

Katz and Weise first considered interactions between *futures* and `call-with-current-continuation` [16]. Futures let a computation be forked in a separate thread and let consumers of the computation receive its value later. A key issue is what it means to resume a continuation inside a future, which is reminiscent of issues we address with resuming continuations inside transactions. However, one typically uses futures for pure computations (though later work considered how to roll back side effects inside futures [27, 22]), making idioms for transactions quite different.

Transactional memory is fundamentally a synchronization mechanism for shared-memory concurrency, but that is not to dismiss message-passing as an alternative. Indeed, the kill-safe synchronization abstractions [8] in DrScheme build on top of Concurrent ML [23] to support robust programming patterns. Conversely, they are little help in preventing shared-memory errors or providing robust synchronization when shared memory is more convenient.

3. Basic Approach

In this section we present a short overview of the operation of atomic transactions, and explain the basic behavior of our system. While at a high level little in this section is novel (in particular, much is shared with AtomCaml [24]), it describes the underlying design and implementation on which we can provide support for continuations and integrate with the Scheme48 interpreter.

3.1 Design

Atomic transactions are introduced via the first-class procedure *atomic*, which takes a thunk, executes it and all its callees atomically, and returns its result. All other threads in the system are prevented from detecting any changes made to global memory by the code protected by the *atomic* call (and vice-versa) until the call returns, at which point the changes are *committed* to the global state.

Without considering the effects of `call-with-current-continuation` (hereafter: `call/cc`) or *retry* (discussed below), *atomic* could be implemented as a procedure like this:

```
(define (atomic f)
  (start-atomic-mode)
  (let ((v (f)))
    (commit-atomic-changes)
    v))
```

`start-atomic-mode` and `commit-atomic-changes` are lower-level transaction-management procedures. If atomic calls are nested, then successive calls to `start-atomic-mode` in the same transaction increment a counter tracking the nesting depth; this counter is decremented by `commit-atomic-changes` until it reaches zero, at which point the entire nested set of atomic calls are committed.

While transactions intend to commit their effects to the global state eventually, it is possible within a transaction to detect a condition suggesting that it should wait until a different global state is present before conducting its computation. A *retry* operation discards any state modifications made thus far in the atomic transaction, and suspends the current thread to wait for other threads to modify shared memory. When the thread in the atomic transaction is resumed, the transaction is restarted from the beginning. The following example naturally involves retrying an atomic operation:

```
(define (atomic-dequeue! q)
  (atomic (lambda ()
    (if (is-empty? q)
        (retry) ; wait for enqueue first
        (dequeue-next! q)))))
```

Implementing *retry* requires more involved control in the *atomic* function than suggested above, but these changes are subsumed by the actual implementation of the full system.

The situation for I/O operations is more complicated than manipulation of heap memory; in general, buffered I/O operations are delayed until commit time, while unbuffered I/O and synchronous message-passing operations are treated as dynamic errors. (Our actual implementation currently does not yet provide this behavior, though it would be a straightforward addition.)

The design of an atomic transaction system must also preserve two additional properties. First, simultaneous execution of two transactions that are not in contention for the same memory locations should not impose a serialization order on the threads. The uniprocessor model described in Section 3.2 obviates this concern in our implementation. Second, an atomic transaction system must provide a notion of fairness, ensuring that a long-running or possibly divergent transaction does not prevent progress by other threads. Our scheduler, described below, ensures this property.

3.2 Implementation

Since Scheme48 multiplexes all user-defined threads on top of a single kernel thread, true simultaneous multiprocessing is impossible. We exploit this behavior by writing atomic changes directly to the affected memory addresses and logging their old values on the side. If a transaction completes before the thread is preempted, the log is discarded. Otherwise, the system is paused to replay the log of old values, restoring the previous program state before running the next thread. The log contains the prior value of all addresses to which a non-initialization write is performed in a transaction. Initialization writes need not be logged as the garbage collector will discard any memory cells allocated in a transaction after references to these new locations are discarded by rollback. A key advantage of the uniprocessor model is that we need not log any read operations or any operations whatsoever outside of transactions. This keeps the overhead in mostly-functional code low. A disadvantage of this technique is that we do not record the set of locations read during a transaction, preventing a demand-driven retry policy. We retry transactions without regard for the particular shared memory addresses modified by other threads in the interim.

A challenge for transactional memory implementations is allowing procedure calls within an atomic scope and determining when writes should be logged without incurring a performance overhead with every operation. Our implementation modifies the runtime system to allow the `start-atomic-mode` primitive to direct the interpreter to use a second opcode dispatch table, which overrides the behavior of heap update operations with logging equivalents. It also installs a fresh rollback log at this time. The `commit-atomic-changes` primitive simply discards this log and restores the original opcode dispatch table. This contrasts with the compilation strategy of compiling each procedure twice—once with code for logging all updates and once with no logging—and changing closures to have two code pointers.

As stated earlier, preemption of an atomic transaction causes it to undo its memory changes before yielding to another thread. If an atomic transaction is not complete after using an entire timeslice, it may need a longer uninterrupted span in which to perform its operations. The next time the transaction is scheduled to run, its quantum is extended to give it more time to complete the entire transaction, which is re-executed from the beginning. While individual timeslices for a transaction increase in length, the frequency with which the transaction is scheduled decreases proportionally. This ensures that our scheduler treats all threads fairly.

4. Patterns for Escaping Transactions

The primary question addressed by this work is what to do when the invocation of a continuation causes control flow to cross the boundary of an atomic call's dynamic extent. While previous work only considered exceptions (i.e., escape continuations), the generic nature of `call/cc` in Scheme requires a more flexible palette of escape behaviors be made available to programmers. For example, `call/cc` can be used to implement exceptions, value consumers,

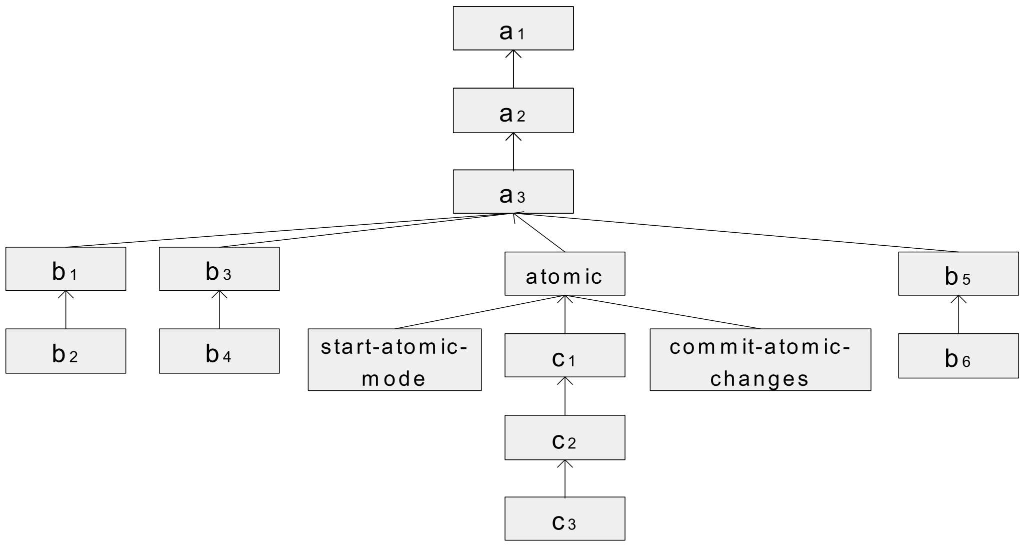

Figure 1. A tree of call stacks used to describe escape behaviors throughout Section 4

and coroutines. Because it is impossible to determine *a priori* which of these patterns a programmer is employing in a given situation, we must provide efficient support for the programmer to choose the appropriate behavior on a case-by-case basis.

In this section, we describe several ways in which transactions and continuations can interact with one another. Each subsection provides a motivating example from Scheme, defines and names the appropriate escape behavior, and gives its intuition both visually and with Scheme code that relies on the low-level atomic primitives first used in Section 3.1.

We demonstrate each meaning using the function call sequence illustrated in Figure 1. Each box in the image represents a procedure call. Arrows point upward from a callee to its caller. Procedure calls on the left precede calls on the right; e.g., procedure b_1 was called and returned before b_3. Depth-wise, procedures return in LIFO order (e.g., b_1 begins before b_2, and b_2 returns before b_1). Naturally, continuations saved with `call/cc` can be used to move freely among available past call-stack states in any order.

If a continuation is both reified with `call/cc` and invoked inside a single call to *atomic* (e.g., c_3 to c_1 or vice-versa), control flows as expected: the transaction continues at the continuation target. Similarly, if a first-class continuation containing a call to *atomic* (b_5 to a_3) is invoked, the subsequent atomic call is performed as described earlier: a separate transaction is used for each invocation of the atomic call.

4.1 Escape as Normal Return

A procedure may use an escape continuation to return values or cede control before its lexical end. In this case, the continuation is being used as a return method, which should act like the lexical end of the atomic call–committing the effects of the atomic computation. For example, consider the following atomic transaction whose body is a loop:

```
(call/cc (lambda (k)
  (atomic (lambda ()
    (let loop ((i 0))
      (start-iteration i)
      (check-for-completion k i)
      (finish-iteration i)
      (loop (+ i 1)))))))
```

Here, the continuation is used as a "break" construct, letting the programmer return a value from `check-for-completion` in the middle of the loop body without executing the `finish-iteration` procedure. Presumably, the programmer returning a value through k wishes to retain the results of the loop computation even though the lexical end of the *atomic* is never reached.

Another example makes the necessity of this behavior more explicit:

```
(lambda (return x)
  (atomic (lambda ()
    (calculate-and-update-list! x)
    (if (good-enough? x)
        (return x))
    (compute-another-list! x)
    (return x))))
```

The `return` continuation delivers the result x to a calling method. While this example uses `return` directly, more realistic code might dynamically nest the call to `return` inside one or more procedure calls. Because the change to x must be permanent in the code above, we must commit the atomic call, even though execution does not reach its lexical end. Therefore, the desired behavior on escape is to commit computation effects and transfer control to the continuation target. We call this behavior "commit-on-escape."

Visual intuition: In Figure 1, a commit-on-escape action would occur if a continuation were saved in any procedure a_i, and a procedure called in the atomic context (any c_i) invoked the continuation to return to a shallower call stack. Because we do not differentiate between escape continuations and continuations used for any other purpose, the transaction is also committed if a continuation into any b_i were invoked within the dynamic extent of *atomic*.

High-level implementation: *atomic* with commit-on-escape can be conveniently implemented using Scheme's `dynamic-wind` procedure as follows:

```
(define (atomic f)
  (dynamic-wind
    (lambda () #f)
    (lambda ()
      (start-atomic-mode)
      (f))
    (lambda () (commit-atomic-changes))))
```

4.2 Escape due to Unexpected Memory State

Continuations are often used by programmers to transfer control to an error handler if the memory state prevents a certain operation.

Consider a non-atomic dequeue operation:

```
(define (get-from-queue! q)
  (if (is-empty? q)
      (queue-empty-failure #f)
      (dequeue! q)))
```

In this case, `queue-empty-failure` is a continuation for an exception handler. If the user implemented a function `get-from-queue-atomically!`:

```
(define (get-from-queue-atomically! q)
  (atomic (get-from-queue! q)))
```

then invocation of the exception handler would exit the dynamic extent of the atomic transaction. Since this is a recoverable error condition—allowing the dequeue merely requires us to wait for another thread to repopulate the queue—it may be correct behavior to wait and retry the operation. Therefore, the invocation of the continuation is merely a signal that a retry operation should occur. A behavior called "rollback-and-retry-on-escape" provides exactly this: the data changes of the atomic transaction are rolled back. The continuation target is replaced by a continuation to the beginning of the atomic call, allowing the call to re-execute from the beginning after memory state changes are made by another thread.

Visual intuition: To visualize this behavior, we refer to Figure 1. Much like with commit-on-escape, the escape behavior occurs if any c_i invokes a continuation into any a_i or b_i. Whereas commit-on-escape first commits the memory changes and then follows the continuation to its target, rollback-and-retry-on-escape undoes any memory changes made in the c_i's and then replaces the continuation's destination with a new target: the beginning of the call to *atomic*. Of course, this will cause an infinite loop unless another thread modifies global state in the meantime, allowing the atomic call to follow a different control path.

High-level implementation: This behavior can be implemented in Scheme as follows:[3]

```
(define (atomic f)
  (let ((retry  (call/cc (lambda (k) k)))
        (should-commit #f))
    (dynamic-wind
      (lambda () #f)
      (lambda ()
        (start-atomic-mode)
        (let ((v (f)))
          (set! should-commit #t)
          v))
      (lambda ()
        (if should-commit
            (commit-atomic-changes)
            (begin
              (rollback-changes)
              (retry retry)))))))
```

[3] R^{5}RS declares that invoking a continuation in an *after* thunk of a `dynamic-wind` call is an undefined operation. Nonetheless, this code compactly expresses the notion we are trying to express with rollback-and-retry-on-escape; the after thunk "redirects" the control transfer.

4.3 Escape as Unrecoverable Panic

The previous subsection addressed exceptional conditions based on shared state, but exceptional control transfers also accompany assertion failures that would not resolve themselves simply by delaying execution of a transaction. Handling these conditions may involve rolling back the data state of the atomic call when its dynamic extent is breached, but following through with the control transfer to the error handler identified by the invoked continuation.

Consider the following example:

```
(define (divide-lists! numer denom)
  (let ((n (car numer))
        (d (car denom)))
    (if (= 0 d)
        (div-zero-failure #f)) ; escape
    (set-car! numer (/ n d))
    (if (not (null? (cdr numer)))
        (divide-lists! (cdr numer)
          (cdr denom)))))

(define (divide-atomically! numer denom)
  (atomic (lambda ()
    (divide-lists! numer denom))))
```

In `divide-lists!`, the lists are divided element-wise and imperatively updated. If an element of the denominator list is zero, an exception handler is invoked via a continuation that escapes the transaction. At this point several memory cells may have already been updated; for the program to continue in a consistent manner, it may need to undo the changes that occurred. But if the denominator list is thread-local, then there is no chance that another thread will update the problematic element, meaning the rollback-and-retry-on-escape behavior would cause an infinite loop.

What is necessary is an escape behavior that undoes memory changes made in the transaction, but does not immediately retry the *atomic* call. This provides a straightforward way of cleaning up the side-effects of a failed computation without requiring programmers to implement case-specific "undo" behavior, eliminating another place for bugs to be introduced.[4] We call this behavior "rollback-and-abort-on-escape." It is similar to a mechanism suggested in Shinnar et al. [25], which provides a control-transferring data-rollback error-recovery mechanism.

This behavior raises two semantic questions. First, why is a control transfer made from a procedure execution that semantically "never happened?" Second, what memory state should be reflected in the thrown value? A programmer may reconcile the first question as she would an interrupt handler: while the trace that caused an exceptional condition to occur is unknown, it is bound to a particular thread, which can be restarted at a known-good reentry point after the exception is handled. With respect to the second, several options are possible. Shinnar et al. make a deep copy of the thrown value object from the abort-time heap state into the globally visible heap, protecting the object's changes from rollback. Fortress does not roll back changes to memory allocated within the transaction. This protects the thrown exception object from rollback, though references from this object to objects allocated pre-transaction will reflect their original memory states. We suggest a third alternative: our implementation rolls back all non-initialization writes to memory made during the transaction. What escapes is the pre-transaction value of the thrown object, or its original value if it was constructed in the transaction. Notice code that throws an immediate value such as `#f`, as `divide-lists!` does, is not affected by this dilemma.

[4] More powerfully, it means undo routines need not be written in an unintuitive LIFO style to consider how far along in the process a multistep mutation has progressed; the transaction log inherently captures this notion.

Visual intuition: Referring to Figure 1, rollback-and-abort-on-escape looks much like commit-on-escape; it occurs when any c_i invokes a continuation into any a_i or b_i, and allows control to be transferred to the continuation's target. But whereas commit-on-escape triggers a commit action upon leaving the dynamic extent of *atomic*, rollback and abort will play the undo log at this time, canceling the transaction's effects on memory.

High-level implementation: This escape behavior could be encoded in Scheme like this:

```
(define (atomic f)
  (let ((should-commit  #f))
    (dynamic-wind
      (lambda () #f)
      (lambda ()
        (start-atomic-mode)
        (let ((v (f)))
          (set! should-commit #t)
          v))
      (lambda ()
        (if should-commit
            (commit-atomic-changes)
            (rollback-changes))))))
```

4.4 Escape and Resumption of an *atomic*'s Extent

`call/cc` can be used to implement more complicated control mechanisms than simple escape targets. One practical example is the use of coroutines for iteration.

Figure 2 shows a program that includes a coroutine iterator over lists. A call to `list-iterator` will initialize an iterator procedure over its argument. Calls to that procedure will return a pair containing the next element of the list, and another iterator over the remainder of the list—or `#f` if the end of the list is reached. The `iterate-until` procedure in this figure uses such an iterator to atomically evaluate whether the elements of a collection sum to a value greater than a certain threshold, and sets a flag if this is the case. Note that the example invocation of `iterate-until` uses the iterator to skip the first element of the list.

In this example, we expose an "overly-complicated" list iterator that uses continuations to yield values and resume execution; normally this acrobatic use of `call/cc` would be hidden behind a library, but it is exposed for the sake of example. A more compelling use of `call/cc` is when iterating over trees or other structures that require backtracking to reach all elements. We present a binary tree iterator and a list iterator built without `call/cc` in Figure 6 in the Appendix to demonstrate other procedures that obey the iterator interface used in Figure 2; `iterate-until` should perform identically regardless of which of these it is passed.

To `iterate-until`, the iterator continuation looks like any other procedure. But because the iterator may be the result of invoking another iterator earlier (as in the example), control may flow outside the dynamic extent of the atomic call when performing iteration. Because `iterate-until` intends for the entire computation over the iterated collection to be performed atomically, the execution inside the body of the iterator must also be encompassed by the atomic transaction, even though it is not within the atomic call's dynamic extent.

To facilitate this, the call *(atomic f)* should switch the Scheme interpreter into a transaction, which remains live until control reaches the end of the call to f, at which point the transaction completes. Escapes from the dynamic extent of f into other dynamic extents are thus the programmer's responsibility to pair with a re-entry into the dynamic extent of f to complete the atomic call.

```
(define threshold-reached #f)

(define (iterate-until threshold iter)
  (atomic (lambda ()
    (set! threshold-reached threshold)
    (let loop ((elt-and-next-fn  (iter)))
      (set! threshold-reached
        (- threshold-reached
          (car elt-and-next-fn)))
      (if (<= threshold-reached 0)
          (set! threshold-reached #t)
          (if (cdr elt-and-next-fn)
              (loop ((cdr elt-and-next-fn)))
              (set! threshold-reached #f)))))))

(define (get-next-elt lst)
  (call/cc (lambda (ret)
    (let loop ((ret ret)
               (lst lst))
      (loop (call/cc (lambda (k)
        (if (eq? (cdr lst) '())
            (ret (cons (car lst) #f))
            (ret (cons (car lst)
              (lambda () (call/cc
                (lambda (ret) (k ret)))))))))
        (cdr lst))))))

; returns an iterator over lst or #f if empty.
(define (list-iterator lst)
  (if (null? lst)
      #f
      (lambda () (get-next-elt lst))))

; example execution, skipping first element
(iterate-until some-threshold
         (cdr ((list-iterator some-lst))))
```

Figure 2. A list iterator implemented as a coroutine and called inside an atomic block requires extend-on-escape semantics

Here, the atomic call's behavior, which is referred to as "extend-on-escape," permits control transfers across the boundary of its dynamic extent without changing the transaction's state.

Visual intuition: In terms of Figure 1, the example code in Figure 2 does the following: The call to `list-iterator` corresponds to procedure b_1. This returns a procedure that iterates the first element, which is immediately called (b_3). The result of this call is a pair containing the first element of the list, and the next iterator. The call to `iterate-until` starts the atomic column of c_i's, but each time the next iterator is invoked (by calling `iter` or `(cdr elt-and-next-fn)`), control leaves column c for a continuation in the dynamic extent of b_3, while remaining in the transaction. This continuation then returns the next *(value . iter)* pair to a continuation in the dynamic extent of `iterate-until` (i.e., a c_i), where the iteration loop continues. The transaction does not end until the call to *atomic* inside `iterate-until` returns.

High-level implementation: The implementation of *atomic* that naturally facilitates this behavior is the simple one initially suggested in Section 3.1:

```
(define (atomic f)
  (start-atomic-mode)
  (let ((v (f)))
    (commit-atomic-changes)
    v))
```

This code may misleadingly suggest that it is acceptable to commit a transaction by using `call/cc` to return to the end of a different

```
(define k #f)

(define (increment-pair-atomically! p)
  (atomic (lambda ()
    (set-car! p (+ 1 (car p)))
    (call/cc
      (lambda (cont) (set! k cont)))
    (set-cdr! p (+ 1 (cdr p))))))

(define (weird-behavior)
  (let ((my-pair  (cons 10 20))
        (done     #f))
    (increment-pair-atomically!
         my-pair)   ; (10 . 20) -> (11 . 21)
    (if (not done)
        (begin
          (set! done #t)
          (k #f)))  ; (11 . 21) -> (11 . 22)
                    ; (causes dynamic error)
    my-pair))
```

Figure 3. Resumption of a completed atomic call breaks the atomic abstraction. It is a dynamic error in our implementation to reach the end of the atomic call a second time.

call to *atomic* than the one used to begin the transaction, which is not the case. A full discussion of nested atomic transactions and the interaction between atomic transactions opened by coroutines is given in Section 5.2.

4.5 Resumption of a Completed Transaction

If a transaction is committed or aborted, it is said to be *completed*; the control state associated with the transaction itself no longer exists. If a programmer has saved a continuation somewhere in the dynamic extent of this transaction, then he may wish to *resume* the completed transaction.

This is problematic from a semantic point of view, because it allows a "fraction" of an atomic transaction to be re-executed—suggesting that transactions are not as atomic as intended. (See Figure 3.) While a first reaction to this problem may be to disallow resumption of completed transactions, this causes many desirable programs to terminate with errors. For example, iterators such as the one in Figure 2 may reside in the dynamic extent of an atomic call. With respect to Figure 2, this would be the case if we replaced the last two lines with:

```
(iterate-until some-threshold
  (list-iterator some-list))
```

If only a portion of the collection was iterated over, then a later transaction may resume iteration of the remaining elements by using a saved iterator. But this requires reentry into the first (completed) transaction's dynamic extent, which would cause an error, even though we intend to use only functional code. On the other hand, reaching the commit action at the lexical end of a completed transaction is problematic because the commit action is no longer paired with the start of a transaction.

The semantics we implemented allow resumption of a completed transaction, but make committing an already-completed transaction a dynamic error. More precisely, control may reenter the dynamic extent of a completed transaction via a continuation, but it must leave the dynamic extent via another continuation before the transaction's lexical end. Reaching the lexical end of a transaction causes the dynamic error. The programmer should also take note that because the transaction has already been completed, side effects in its dynamic extent are no longer performed atomically; i.e., it is not part of a transaction unless another transaction is open at the time of resumption.

Visual intuition: In terms of Figure 1, resumption occurs in the following circumstances. A continuation to some c_i is saved during the execution of the transaction, which is then committed. Later (e.g., b_5 or b_6), this continuation is invoked, shifting control back into the dynamic extent of the *atomic* call, even though the code is not being executed atomically. Allowing c_1 to then return to its enclosing *atomic* is a dynamic error, but if a c_i escapes the transaction's extent via another continuation (e.g., back into b_5) then execution proceeds as expected.

High-level implementation: Supporting resumption while preventing recommits of a completed atomic call can be accomplished with the following implementation of *atomic*:

```
(define (atomic f)
  (let ((closed  #t))
    (start-atomic-mode)
    (set! closed #f)
    (lambda ()
      (let ((v  (f)))
        (if closed
            (error "Double commit"))
        (set! closed #t)
        (commit-atomic-changes)
        v))))
```

5. Semantics For Transactions in Scheme

Where Section 4 described use cases where different escape behaviors are required, here we describe the particular procedures we added to Scheme to provide atomic transactions supporting each of these situations.

In the previous section, the escape behavior was defined by changing the implementation of the *atomic* procedure. But there are actually three points in time when it makes sense to specify the intended behavior:

1. When reifying a continuation
2. When opening an atomic call
3. When invoking the continuation

What follows is a description of procedures targeting these different places in the code, and an informal semantics explaining how they interact when used together. Table 1 lists all relevant procedures, providing both long- and short-form names as well as their types.

5.1 Individual Behaviors

In the following paragraphs, we describe the behavior of each category of procedures from Table 1 when used in isolation.

The call/cc-* family: The *call/cc-** procedures in Table 1 operate like the canonical `call/cc`; they reify the current continuation and pass it as an argument to the lambda passed to `call/cc`. The difference is that the continuation has an additional property attached that specifies the behavior to use when invoking this continuation escapes an atomic call. The attached behavior is ignored if invoking the continuation does not escape an atomic call.

A user can also redefine the escape behavior bound to a particular continuation with *set-continuation-escape-behavior!*, which takes a continuation and a value indicating which behavior to use. A companion procedure, *continuation-escape-behavior*, retrieves the behavior associated with a continuation.[5]

[5] Actually, Scheme48 continuations are returned wrapped in procedures, preventing direct manipulation. We could achieve the described functional-

Reifying continuations:		
call/cc-with-commit-on-use	call/cc/commit	$((\alpha \to void) \to \alpha) \to \alpha$
call/cc-with-rollback/abort-on-use	call/cc/rollback	$((\alpha \to void) \to \alpha) \to \alpha$
call/cc-with-rollback/retry-on-use	call/cc/retry	$((\alpha \to void) \to \alpha) \to \alpha$
call/cc-extends-on-use	call/cc/extend	$((\alpha \to void) \to \alpha) \to \alpha$
call/cc-defers-behavior	call/cc/defer	$((\alpha \to void) \to \alpha) \to \alpha$
Managing continuation-bound behavior:		
set-continuation-escape-behavior!		$(\alpha \to void) \times behavior \to unspecified$
continuation-escape-behavior		$(\alpha \to void) \to behavior$
Defining atomic calls:		
atomic-with-commit-on-escape	atomic/commit, atomic	$(\to \alpha) \to \alpha$
atomic-with-rollback/abort-on-escape	atomic/rollback	$(\to \alpha) \to \alpha$
atomic-with-rollback/retry-on-escape	atomic/retry	$(\to \alpha) \to \alpha$
atomic-extends-on-escape	atomic/extend	$(\to \alpha) \to \alpha$
Continuation invocation point-bound behavior:		
escape-and-commit		$(\alpha \to void) \times \alpha \to void$
escape-and-rollback/abort		$(\alpha \to void) \times \alpha \to void$
escape-and-extend		$(\alpha \to void) \times \alpha \to void$
Explicitly rolling back transactions:		
rollback/abort		$\to void$
rollback/retry	retry	$\to void$

Table 1. The procedures our library provides to facilitate atomic transactions

Atomic call definitions: The atomic call defines the boundaries of a transaction and can specify the behavior to be used by a continuation when escaping its dynamic extent. Several variants of the *atomic* procedure are provided, each of which is semantically equivalent to an implementation suggested in Section 4. *atomic* itself is a synonym for *atomic-with-commit-on-escape*. If a continuation created with `call/cc` or *call/cc-defers-behavior* is used to escape the dynamic extent of such an atomic call, the atomic call defines the escape behavior.

Call-site specific behavior: There may be occasions when programmers wish to override all other precedence rules and force a particular behavior for a given continuation invocation. The commands *escape-and-commit*, *escape-and-rollback/abort*, and *escape-and-extend* each receive as arguments a continuation and a value to pass to the continuation; the continuation is applied to the value and the behavior specified by the procedure is used, regardless of the behavior bound to the continuation or the enclosing atomic call. An unfortunate property of these procedures is that they break the continuations-as-procedures abstraction otherwise maintained in Scheme. Nonetheless, they are a straightforward mechanism providing specific behavior in locations where it might be otherwise cumbersome to achieve.

Explicit rollback commands: Our library provides programmers with two explicit rollback commands to abort the current transaction: *rollback/retry* and *rollback/abort*. While both complete all live atomic calls by undoing all memory operations since the first live atomic call began, the post-rollback control flow differs between these two commands. *rollback/retry* resets control to the beginning of the first atomic call in the current transaction and begins re-executing the transaction. *rollback/abort* treats the first atomic call made as a `call/cc` that has just returned; control is sent to the end of this initial atomic call.

ity via a lookup table with weak pointers to the continuations, but for expedience *call/cc-** actually take as an argument a two-parameter procedure—the continuation k and a handle h. This handle is then used to manipulate the escape behavior for its companion continuation. The original `call/cc` form remains unmodified, silently discarding its handle.

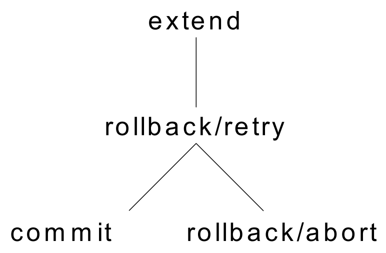

Figure 4. Nested escape behaviors form a semilattice

5.2 Nested and Conflicting Behaviors

The problem with letting the programmer specify the behavior of escaping continuations is that multiple specifications can conflict. First, invoking a continuation can cause control to transfer across the dynamic extent of multiple atomic calls and these calls may have specified different behaviors. Second, an atomic call and a *call/cc-** call might specify different behaviors. In this section we discuss how we resolve both kinds of conflicts.

Nested atomic calls: Calls to the *atomic-** procedures can be nested both statically and dynamically. Maintaining this capability is important to preserve the opacity of library calls.

When all nested atomic calls have the same escape behavior, the common escape behavior is used as a continuation escapes from the entire set of atomic calls. (For example, if several atomic calls were opened inside one another, each created by *atomic-with-commit-on-escape*, escape from this set of atomic calls would commit them all.) If these calls were instead made with different *atomic* derivatives, then this requires a more complicated rule. It is impossible, for example, to satisfy both rollback/abort and commit operations simultaneously. While our current implementation simply raises a dynamic error, a more robust solution is to arrange the four escape behaviors in the semilattice shown in Figure 4. When escaping from a set of nested atomic calls, the escape behaviors associated with these calls are collected and joined to find their least upper bound; this behavior is then applied to all the atomic calls being escaped.

We chose the hierarchy described in Figure 4 because rollback/retry is in general always "safer" than commit or rollback/abort. It will never lose a computation, nor will it inadvertently commit undesired changes. The *extend* behavior sits at the top of the lattice because programmers who expressly intend to leave and resume the dynamic extent of a transaction must ensure that all nested transactions are closed properly.

Atomic call behavior vs. call/cc behavior: The situation is further complicated by adding continuation-bound escape behaviors to the mix. A programmer can reify a continuation using one of the procedures in the first section of Table 1, such as *call/cc-with-commit-on-use*, and then use it to escape a transaction opened with (for instance) *atomic-with-rollback/retry-on-escape*. Our rule is that the behavior associated with the continuation overrides the escape behavior associated with the atomic call itself. The behavior lattice is ignored in this case, as it often makes sense to allow multiple distinct escape behaviors in the same transaction.

We believe that this hierarchy makes the most sense when bringing together multiple behavior specifications. Consider the following example procedure:

```
(define (yield-from-queue! return q)
  (atomic (lambda ()
    (return (get-from-queue! q)))))
```

This procedure invokes a continuation named `return` to yield values from the procedure body and should commit its enclosing transaction. By contrast, the `get-from-queue!` procedure defined in Section 4.2 may invoke an empty-queue-handling continuation which, in an atomic transaction, should be ignored in favor of retrying the transaction. Even though these behaviors conflict with one another, this procedure can be implemented by defining `yield-from-queue!` to commit on escape (by using *atomic-with-commit-on-escape*), but attaching the rollback-and-retry-on-escape behavior to the error-handler continuation invoked in `get-from-queue!` by reifying the continuation with *call/cc-with-rollback/retry-on-use*.

There may be cases in which the ability to override the call/cc-driven behavior with the behavior specified by the atomic call is desirable, but we believe that the precedence we have implemented makes sense in the majority of cases. There is fundamentally an "arms race" between clients of continuations and definitions of continuations to provide the definitive behavior. Our implementation allows the continuation to override the atomic call, while noting that instrumentation of the continuation call-site with one of the *escape-and-** procedures provides the atomic call with a "last chance" mechanism to mandate behavior.

Coroutine transactions: By using the extend-on-escape behavior, it is possible to move between live atomic calls and close them in any order, implying that one is not properly nested within the other. This is possible in cases involving mutually-recursive coroutines containing atomic calls. Referring back to Figure 1, a_3 can create an atomic call (the column of c_i's), which then escapes back into a_3. Another atomic call (e.g., another column of c_i's parallel to the existing one, but not shown) represents the body of an additional coroutine, which may invoke continuations into the original c_i column and vice-versa. The coroutines will be on different dynamic extents, each containing a distinct atomic call. Because they use the extend-on-escape behavior, these atomic calls remain live even when leaving one dynamic extent for the dynamic extent of the other coroutine via a continuation.

The result of this activity is atomic calls that are interleaved rather than nested; there is no "inner" or "outer" atomic call. In Figure 5, the upper diagram represents atomic calls which are nested in dynamic extent. The "inner" atomic calls are committed

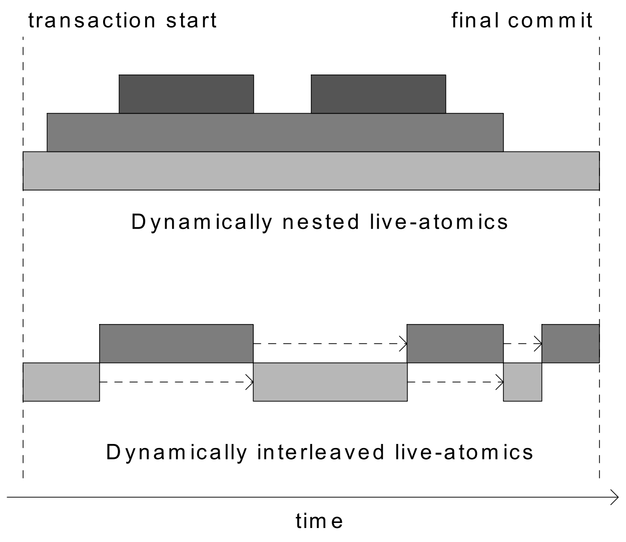
transaction start
final commit
Dynamically nested live-atomics
Dynamically interleaved live-atomics
time

Figure 5. A comparison of LIFO-structured transactions and coroutine transactions. Boxes represent the atomic call(s) live on the current dynamic extent at a point in time; different shades represent different calls to *atomic*.

before the outermost atomic call commits; the last commit actually makes the memory changes available to other threads. In the lower diagram, an atomic call is suspended with the extend-on-escape behavior and another atomic call is begun. The "dormant" atomic call remains live, as denoted by the dashed arrows, and will be resumed later by a continuation. These atomic calls continue to transfer control between one another until they are all individually committed, at which point memory changes are made visible to other threads. Because LIFO order is not maintained internally, they need not be committed in LIFO order. Of course, interleaved atomic calls may make additional LIFO-structured atomic calls as well, though this is not shown in Figure 5.

In cases where a continuation escapes one or more atomic calls on the current dynamic extent, the key question is whether it also implicitly escapes the dormant live atomic calls on other dynamic extents. In our system, the answer is no; escape behavior is determined by considering only what happens as a continuation escapes atomic calls on the current dynamic extent. To commit the total transaction, all atomic calls on all dynamic extents must be committed either by escaping properly from each of the calls or by reaching their respective lexical endpoints. The order in which the atomic calls are completed is insignificant.

Contrary to the commit operation, rollback in the form of retry or abort affects the entire transaction containing all live atomic calls, even if individual nested or interleaved components have been committed. Therefore, if a continuation is used to escape an atomic call with the rollback-and-abort-on-escape behavior, the continuation must escape from all live atomic calls; if the continuation target is inside another live atomic call, it is a dynamic error. It is tempting to attempt to support partial abort or retry operations such as described in [2], but this existing work presupposes that atomic calls are nested and commit in LIFO order. Because of the potential for atomic calls to become interleaved, it is not clear which memory operations should be undone by a partial rollback operation or where a transaction should be restarted from in the event of a partial retry operation.

Implementing our semantics in the presence of any number of nested and interleaved atomic calls is refreshingly straightforward.

When a transaction is not running, invoking *atomic-** begins a transaction. Subsequent invocations of *atomic-** cause the thread's live atomic call count to increase by one. The total transaction does not commit until every live atomic call is completed. If the live atomic call counter is greater than one, committing an atomic call simply decrements the counter and marks the particular call as completed. The final commit operation sets the counter to zero and publishes the memory changes from the transaction to the global state.

6. Implementation

This section describes our implementation, which has two components. The first is a set of modifications to the run-time system of Scheme48 to facilitate atomicity on top of its existing memory model. The second is a library of procedures called by the user to denote blocks of code to be run in atomic transactions, and to create continuations with particular escape semantics.

Our implementation is relatively compact; we added roughly 1,000 lines of code to an 80,000 line code-base, and modified about another 500 of the existing lines of code.

We chose the Scheme48 implementation of Scheme to modify for three reasons: it is fully $\mathrm{R}^5\mathrm{RS}$ compliant, it has been used successfully in projects before, and it was designed to be easily understood and modified [18].

In the rest of this section, we describe the modifications to the run-time system in greater detail, and the ramifications of these modifications on the behavior of *rollback/abort*. This is followed by a description of our additions to the user-accessible library. We then evaluate our implementation in terms of asymptotic complexity and performance on some microbenchmarks.

6.1 Run-time System Modifications

As mentioned in Section 3, we took advantage of the threading model used in Scheme48 to simplify the demands of atomicity. When atomic mode is enabled, a log retains information about the old value of a changed memory location in case a rollback event occurs. When the scheduler interrupts a thread in an atomic transaction, it is rolled back before another thread runs. When the atomic thread is resumed (starting again from the beginning of the atomic transaction), the scheduler doubles the length of its timeslice, allowing it to make more progress toward completion. To prevent starving all other threads, as a thread's timeslice is increased, its scheduling frequency is decreased proportionally. When the atomic transaction is committed, the timeslice for the thread is restored to its original duration. Timeslice modification occurs only if the entire previous timeslice was spent executing a transaction, so that threads are not given a timeslice increase unnecessarily.

Scheme48 uses a bytecode compiler and virtual machine to execute programs; Scheme text is compiled to bytecode operations that are then dispatched by an array of procedures that perform one high-level instruction each. Our implementation doubles the length of this array and uses it in two logical halves. Procedures $0 \ldots n-1$ represent the normal "non-atomic" operations. Procedures $n \ldots 2n-1$ are their atomic-mode equivalents. Most opcodes reuse the same procedure in both halves. Certain opcodes, such as those that govern non-initialization memory updates, use a different procedure in the atomic half of the array to insert an entry into a log containing the previous value of the memory address modified, in addition to performing the normal memory update operation. By doubling the number of opcodes and maintaining in the virtual-machine state which opcodes to use, we incur no per-instruction overhead and do not need to recompile any code before it can be used within a transaction.

Additional opcodes were added to switch the half of the array the opcode dispatch routine uses, and to perform commit and rollback operations. The rollback opcode replays the rollback log in LIFO order, restoring the initial memory state. The commit opcode discards the rollback log.

6.2 Information Escape on Rollback

Because of the particular logging policy we implemented, invocation of a continuation with the rollback-and-abort-on-escape behavior can cause potentially surprising results if programmers do not carefully consider the values passed to the continuation. Immediate values (`42`, `#t`, `'()`, etc.) are always transmitted without error. But any mutable data structures promoted out of the atomic transaction will have any memory mutations rolled back, despite their escaping status. Any data structures constructed in the atomic transaction will still survive (the `cons` operation itself is not undone, as superfluous `cons` cells will be garbage collected later), but their mutations are rolled back. Thus the expression:

```
(atomic-with-rollback/abort-on-escape
  (lambda ()
    (let ((foo (cons 1 2)))
      (set-car! foo 100)
      (set-cdr! foo (cons 3 4))
      (k foo)))) ; k is some continuation
```

will return `(1 . 2)` to the continuation reified in k. Programmers are responsible for determining that only "good" data is passed to continuations with the rollback-and-abort-on-escape behavior.

6.3 User Library

Most of Scheme48's functionality is in a large package library containing procedures available to Scheme programs. We added an `atomic` package containing the procedures listed in Table 1.

We extended the structure Scheme48 uses for per-thread state to include data relevant to the thread's transaction state. This data includes the number of live atomic calls, the escape behavior to use for the current dynamic extent, and the continuations to invoke on rollback/abort or retry. As the existing thread structure is already fairly heavyweight, this is in line with the Scheme48 implementation. Some of this information changes based on the innermost atomic call on the current dynamic extent. Thus the thread structure is updated through thunks installed with `dynamic-wind`. These thunks also perform the user's specified escape behavior and prevent multiple commits, as Section 4 described.

Our implementation relies on a Scheme48-specific behavior: while $\mathrm{R}^5\mathrm{RS}$ specifies that invocation of a continuation inside a dynamic-wind *before* or *after* thunk is undefined, Scheme48 follows the continuation to its target in the "usual manner."[6]

6.4 Implementation Complexity

In this subsection we briefly discuss the asymptotic complexity of various operations in our implementation. Most operations incur very little overhead.

Rollback is potentially the most time-consuming operation; for every memory write performed in the atomic transaction, it must be undone. Rollback buffers are $O(n)$ in the number of writes performed, not in the number of locations mutated. This makes logging of mutations $O(1)$ as they simply prepend to the list. We believe this is important because commit is the most common case for

[6] While we exploit this Scheme48 behavior for convenience, we believe it is unnecessary for building such an atomic system. Scheme48's `dynamic-wind` behavior is simply an efficient way to encode our nested escape behavior handling. Without this property, we would have made changes lower down in the Scheme48 virtual machine.

short transactions [24]. This does not always yield the best performance; for long-running transactions that modify a few locations many times, a hashing-based log would prove more efficient.

Atomic commit is an $O(1)$ operation, as it simply discards the root of the rollback log, which is garbage collected later.

Escape from an atomic transaction is $O(n + m)$ in the number of atomic calls open on the escaping dynamic extent (n) and the target dynamic extent (m), because the thunks installed by `dynamic-wind` for each atomic call must be executed. Beginning a transaction is an $O(n)$ operation in the height of the call stack because it must save a continuation, which is an $O(n)$ operation in Scheme48 (though in testing, we discovered that the other constant-time operations involved in beginning a transaction dominate this penalty for most reasonably-sized call stacks). Entering subsequent atomic calls inside the same transaction are $O(1)$ operations. Leaving a single nested atomic call is also an $O(1)$ operation.

The total memory consumed by a transaction is $O(n+m)$ where n is the number of live atomic calls and m is the number of memory mutations performed during the transaction. For most transactions, m will dominate n.

6.5 Evaluation and Microbenchmarks

We tested our implementation against a suite of test programs we wrote to exercise the various aspects of our feature set and confirmed that they operate as expected. All example code provided in this paper operates correctly on our implementation.

We evaluated our system's performance on two microbenchmarks to demonstrate that the semantics we describe is feasible on current hardware without suffering a major performance penalty. The benchmarks are described below, and the results are shown in Table 2. All benchmarks were run in "benchmark mode" in Scheme48, which allows the bytecode compiler to inline procedures for better performance. Tests were conducted on a dual-core Pentium-D system operating at 3.2 GHz with 1 GB of RAM, running Red Hat Fedora Core 5.1 with Linux kernel version 2.6.16. The Scheme48 version modified by our implementation is v1.4.[7]

In one benchmark, we generate a list of n integers, $0 \ldots n-1$. In an atomic transaction, we imperatively increment each list element, resulting in a list ranging over $1 \ldots n$. We compare the performance of this with an unsynchronized version; with one using Scheme48's "proposals" mechanism which provides the most straightforward comparison; and with one which obtains and releases a lock when incrementing each `cons` cell. A "real world" lock-based system would have performance somewhere between that of the completely unlocked case and the individually locked case, with performance inversely proportional to the granularity of locking. It is unclear why the proposal-based case performed so much slower than the other tests. Results are reported in both the number of list elements processed per second (in multiples of 100,000), and as a relative metric scaled such that our *atomic* primitive is 1.00. (Higher values represent better performance.)

We further explored the list processing benchmark space by timing the iterator mechanism in Figures 2 and 6. Lists of integers manipulated through `cons` cells were iterated over with both coroutine and "straightforward" iterator procedures. The coroutine iterator was timed in both an atomic context and in an "unsafe" non-synchronized context. The procedural iterator (Figure 6) was only run atomically. In both cases, coroutines were much slower than the procedural iterator due to the overhead of `call/cc`; performing iteration atomically added a modest overhead of 17%.

[7] While the current version is 1.6, the most recent version available in January 2007 when we began our work was 1.4. The differences between 1.4 and 1.6 are small enough that this should not be a major factor.

	list		matrix	
	10^5 elts/s	scaled	10^3 elts/s	scaled
atomic	24.32	1.00	5.05	1.00
proposal	0.024	0.001	1.42	0.28
unlocked	26.27	1.08	6.15	1.21
locked	1.66	0.07	5.97	1.18
atomic-coroutines	1.20	1.00		
unsafe-coroutines	1.40	1.17		
atomic-procedural	13.27	11.05		

Table 2. Performance Microbenchmarks

In another benchmark, we multiply two $n \times n$ matrices of integers to yield a third $n \times n$ matrix. This was performed using our atomic transactions, with no locking, using Scheme48 proposals, and with locks on each row or column of an input matrix. It is interesting to note that since we used the `arrays` structure defined in SRFI 47 to implement matrices, for the proposal-based benchmark we had to modify the `arrays` package to provide getter and setter functions that used the underlying data structure in a proposal-safe manner, underscoring the lack of abstraction when using explicit proposal-based memory access.

The benchmark results in Table 2 demonstrate that our implementation of atomic transactions provides computation throughput at an acceptable performance rate. The existing Scheme48 proposals mechanism incurs a large performance penalty when used in each benchmark. The atomic transactions have between an 8% and 21% overhead relative to the case with no synchronization. Adding explicit locks to the code causes a sharp performance decrease proportional to the degree of locking granularity.

Further testing demonstrated that our modifications to Scheme48 do not cause a significant performance difference in non-atomic code when compared to the original Scheme48 system.

7. Future Work and Conclusions

The primary limitation of our current implementation is that performing I/O within a transaction is undefined behavior. Buffering output until the transaction commits would be straightforward, but a more thorough solution would require adapting related work [11, 4, 3, 24] on this unsettled issue. Additional future work on the language definition includes a more rigorous semantic definition and considering the interaction between transactions and other control effects (e.g., shift/reset). For the latter, the most relevant to Scheme are the "error" and "exception" constructs described in SRFIs 23 and 34, respectively. (Since the reference implementation described in SRFI 34 is defined in terms of `call/cc`, there may be few new concerns.)

Defining what it should mean when a continuation invocation crosses a transaction boundary is a difficult decision that ultimately depends on both the intended use of the continuation and the nature of the transaction itself. Therefore, we defined a variety of behaviors and demonstrated how to implement them. Much of the difficulty arises from continuations' many uses; one way to view our user-specified escape behaviors is as a way of "taming" `call/cc` by requiring annotations. We know of no other way to provide first-class continuations and software transactions in the same language. Our implementation, utilizing a combination of high-level Scheme techniques (e.g., uses of `dynamic-wind`) and low-level interpreter techniques (e.g., duplicating the opcode table), has good performance and will be a useful tool for investigating programming with software transactions.

References

[1] N. I. Adams IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, J. G. L. Steele, G. J. Sussman, M. Wand, and H. Abelson. Revised^5 report on the algorithmic language Scheme. *SIGPLAN Not.*, 33(9):26–76, 1998.

[2] A.-R. Adl-Tabatabai, B. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *ACM Conference on Programming Language Design and Implementation*, pages 26–37, June 2006.

[3] E. Allen, D. Chase, J. Hallet, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt. The Fortress language specification, version 1.0β, Mar. 2007. http://research.sun.com/projects/plrg/Publications/fortress1.0beta.pdf.

[4] C. Blundell, E. C. Lewis, and M. M. K. Martin. Unrestricted transactional memory: Supporting I/O and system calls within transactions. Technical Report TR-CIS-06-09, Department of Computer and Information Science, Univ. Pennsylvania, May 2006.

[5] B. D. Carlstrom, J. Chung, A. McDonald, H. Chafi, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. In *ACM Conference on Programming Language Design and Implementation*, pages 1–13, June 2006.

[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, 2005.

[7] Cray Inc. Chapel specification 0.4. http://chapel.cs.washington.edu/specification.pdf.

[8] M. Flatt and R. B. Findler. Kill-safe synchronization abstractions. In *ACM Conference on Programming Language Design and Implementation*, pages 47–58, June 2004.

[9] M. Gasbichler and M. Sperber. Integrating user-level threads with processes in Scsh. *Higher-Order and Symbolic Computation*, 18(3–4):327–354, 2005.

[10] N. Haines, D. Kindred, J. G. Morrisett, S. M. Nettles, and J. M. Wing. Composing first-class transactions. *ACM Trans. Prog. Lang. Syst.*, 16(6):1719–1736, 1994.

[11] T. Harris. Exceptions and side-effects in atomic blocks. *Science of Computer Programming*, 58(3):325–343, Dec. 2005.

[12] T. Harris and K. Fraser. Language support for lightweight transactions. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, Oct. 2003.

[13] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *ACM Symposium on Principles and Practice of Parallel Programming*, pages 48–60, June 2005.

[14] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *ACM Conference on Programming Language Design and Implementation*, pages 14–25, June 2006.

[15] B. Hindman and D. Grossman. Atomicity via source-to-source translation. In *ACM SIGPLAN Workshop on Memory Systems Performance & Correctness*, Oct. 2006.

[16] M. Katz and D. Weise. Continuing into the future: On the interaction of futures and first-class continuations. In *ACM Conference on LISP and Functional Programming*, pages 176–184, 1990.

[17] R. Kelsey, J. Rees, and M. Sperber. The incomplete Scheme 48 reference manual for release 1.6, section 7.4, Feb. 2007. http://s48.org/1.6/manual/manual-Z-H-9.html#node_sec_7.4.

[18] R. A. Kelsey and J. A. Rees. A tractable Scheme implementation. *Lisp Symb. Comput.*, 7(4):315–335, 1994.

[19] J. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2006.

[20] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Prog. Lang. Syst.*, 5(3):381–404, 1983.

[21] J. Manson, J. Baker, A. Cunei, S. Jagannathan, M. Prochazka, B. Xin, and J. Vitek. Preemptible atomic regions for real-time Java. In *26th IEEE Real-Time Systems Symposium*, Dec. 2005.

[22] L. Moreau. The semantics of Scheme with future. In *1st ACM International Conference on Functional Programming*, pages 146–156, 1996.

[23] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[24] M. F. Ringenburg and D. Grossman. AtomCaml: First-class atomicity via rollback. In *10th ACM International Conference on Functional Programming*, pages 92–104, Sept. 2005.

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

[26] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in STM. In *ACM Conference on Programming Language Design and Implementation*, June 2007.

[27] P. Tinker and M. Katz. Parallel execution of sequential Scheme with ParaTran. In *ACM Conference on LISP and Functional Programming*, pages 28–39, 1988.

```
;; Binary trees are represented as
;; (val . (left-tree . right-tree))
;; Empty trees are represented as '()
(define (bt-next bt next ret)
  (if (not (null? (cadr bt)))
      (set! ret (call/cc (lambda (k)
        (bt-next (cadr bt) k ret)))))
  (if (not (null? (cddr bt)))
      (let ((new-ret (call/cc (lambda (k)
        (ret (cons (car bt) (lambda ()
          (call/cc (lambda (in) (k in))))))))))
        (bt-next (cddr bt) next new-ret))
      (if next
          (ret (cons (car bt)
            (lambda () (call/cc (lambda (in)
              (next in))))))
          (ret (cons (car bt) #f)))))

(define (tree-iterator tree)
  (if (null? tree)
      #f
      (lambda () (call/cc (lambda (ret)
                    (bt-next tree #f ret))))))

; Another list iterator, without continuations
(define (lst-next lst)
  (if (null? (cdr lst))
      (cons (car lst) #f)
      (cons (car lst)
        (lambda () (lst-next (cdr lst))))))

(define (list-iterator lst)
  (if (null? lst)
      #f
      (lambda () (lst-next lst))))
```

Figure 6. Additional procedures that return iterators conforming to the interface in Figure 2

A. Iterator Examples

Figure 6 presents additional iteration functions conforming to the interface in Figure 2. Both `list-iterator` and `tree-iterator` accept a data structure to iterate over and return a thunk that is an iterator, or `#f` if there are no iterable elements. Calling the thunk returns a pair containing the next value in the collection and another thunk (or `#f`). The `iterate-until` procedure in Figure 2 operates identically regardless of which iterator is used, and regardless of whether the iterator is initially invoked inside a transaction.