Semantics of Transactional Memory and Automatic Mutual Exclusion

Martín Abadi[†⋆] Andrew Birrell[†] Tim Harris[‡] Michael Isard[†]

Microsoft Research, Silicon Valley[†] University of California, Santa Cruz[⋆] Microsoft Research, Cambridge[‡]

abadi@microsoft.com birrell@microsoft.com tharris@microsoft.com misard@microsoft.com

Abstract

Software Transactional Memory (STM) is an attractive basis for the development of language features for concurrent programming. However, the semantics of these features can be delicate and problematic. In this paper we explore the tradeoffs between semantic simplicity, the viability of efficient implementation strategies, and the flexibility of language constructs. Specifically, we develop semantics and type systems for the constructs of the Automatic Mutual Exclusion (AME) programming model; our results apply also to other constructs, such as atomic blocks. With this semantics as a point of reference, we study several implementation strategies. We model STM systems that use in-place update, optimistic concurrency, lazy conflict detection, and roll-back. These strategies are correct only under non-trivial assumptions that we identify and analyze. One important source of errors is that some efficient implementations create dangerous "zombie" computations where a transaction keeps running after experiencing a conflict; the assumptions confine the effects of these computations.

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

General Terms Languages, Theory

1. Introduction

The notorious difficulty of concurrent programming stems in part from the challenges of expressing the intended program semantics with the available constructs for synchronization and mutual exclusion. For example, programs with threads and locks often suffer from deadlocks and race conditions. Some recent type systems and other program analyses aim to prevent these errors (e.g., (Sterling 1993; Abadi et al. 2006; Naik et al. 2006)). More radically, many researchers have been exploring the use of Software Transactional Memory (STM) (Shavit and Touitou 1995) as a basis for language features that may make it easier to develop and analyze concurrent programs. In one approach, locks are replaced with block-structured atomic sections, so a programmer may reason as if each atomic section is executed as a single step, serialized with respect to all other atomic sections (Harris and Fraser 2003; Harris et al. 2005; Carlstrom et al. 2006). Several other related models have been proposed; these include Safe Futures (Welc et al. 2005), Transactions Everywhere (Kuszmaul and Leiserson 2003), and Automatic Mutual Exclusion (AME) (Isard and Birrell 2007), described in Section 2, on which this paper is based.

Intuitively, the semantics of STM is appealingly simple. However, as researchers are coming to discover, this simplicity is illusory and the actual semantics offered by implementations are often counterintuitive—programs that look "obviously correct" may behave in unexpected ways. The crux of the problem is that implementations do not detect conflicts between a transaction running in one thread and non-transactional steps of another thread. This property, sometimes termed "weak atomicity" (Blundell et al. 2005), is attractive from an implementation standpoint: it means that non-transacted code does not incur a performance overhead, and that existing libraries and operating system interfaces can be used without recompilation in non-transacted code. In contrast, "strong atomicity" requires the avoidance or detection of those conflicts. Strong atomicity appears to be the semantics expected by programmers but, unfortunately, it does not appear to be practical to implement using STM without restrictions and without recompiling non-transacted code.

This paper examines this problem and explores the tradeoffs between semantic simplicity, the use of efficient implementation strategies, and the flexibility of language constructs. We present our results focusing on the AME programming model for two reasons. First, while developing this new programming model, we hope to avoid the pitfalls we have encountered with earlier work on atomic blocks; we want to understand AME's constructs and which techniques we can use to implement them. Second, there is a straightforward mechanical translation from a program with atomic blocks into AME's constructs, so the results that we establish will apply more broadly; the translation in the other direction is less obvious.

We present the AME calculus as a formalization of the AME programming model (Section 3) and define a strong semantics for this calculus that abstracts the underlying STM (Section 4). We show that, without language restrictions, the techniques used by practical STMs can lead to behavior that is incorrect under the strong semantics (Section 5). Earlier work has provided some examples (Blundell et al. 2005; Shpeisman et al. 2007). We argue that most of these are incorrectly synchronized programs; however, we show a number of further examples which, informally, do not contain race conditions. We focus, in particular, on the problems that occur when using the Bartok-STM implementation (Harris et al. 2006) in which updates are made in-place to the heap ("eager versioning" (Moore et al. 2006)) so tentative work is visible before a transaction commits, and conflicts may not be detected until commit time ("lazy conflict detection"), allowing a transaction to continue running as a "zombie" (Dice et al. 2006) after becoming conflicted. Similar implementation choices have been made in other

STM systems such as McRT (Saha et al. 2006), because of their efficiency on many practical workloads.

We then examine language restrictions that enable weaker semantics—which model some of the techniques used by Bartok-STM—to implement these examples correctly (Section 6). First, we consider a *violation-freedom* condition, which formalizes the sense in which our examples of Section 5 are race-free. For programs that satisfy this condition, we show that a lower-level semantics with weak atomicity, in-place update, and roll-back implements the strong semantics (Section 7). In this semantics, at most one transaction executes at a time. While this semantics is still some way from an actual implementation, it resembles a practical uni-processor STM (Manson et al. 2005a).

A further language restriction is a type system that statically separates data according to whether or not it is accessed transactionally. We show that, for well-typed programs, a weaker semantics that models the concurrent execution of transactions and lazy conflict detection between them implements the strong semantics (Section 8). Violation-freedom does not suffice for this property.

We discuss related work in Section 9. We conclude in Section 10 by considering further work, and the implications of our results to the implementation choices made within an STM and to the design of language features based on it. Proofs and additional results are available at `http://research.microsoft.com/research/sv/ame/`.

2. Automatic Mutual Exclusion

The AME programming model has been outlined in a workshop paper (Isard and Birrell 2007). We summarize its constructs here, and refer to that paper for supporting details and examples.

2.1 AME Basics

The motivation for AME is to encourage programmers to place as much of the program text inside transactions as possible—we refer to this as "protected" code—leaving non-transacted "unprotected" code primarily for interactions with legacy code. We believe that this "protected by default" style will help programmers write concurrent code whose semantics are clearer than is typical with today's languages; in particular, programs in this style should be easier to understand and to maintain than those with lock-based idioms, or with a straightforward translation of lock-based code to use atomic blocks.

Running an AME program consists of executing a set of asynchronous method calls. The AME system guarantees that the program execution is equivalent to executing each of these calls (or their fragments, defined below) in some serialized order. AME achieves concurrency by overlapping the execution of the calls in cases where they are non-conflicting. The program terminates when all its asynchronous method calls have completed. Initially, the set consists of a call of `main()` initiated by the AME system. As well as ordinary method calls, code can create another asynchronous method call by executing:

```
async MethodName(<method arguments>);
```

The calling code continues immediately after this call. In the conceptual serialization of the program, the asynchronous callee will be executed after the caller has completed.

In order to achieve the serialization guarantee, we envision that each asynchronous method call will be executed by the AME system as a transaction, in a thread provided by the system. If a transaction initiates other asynchronous method calls, their execution is deferred until the initiating transaction commits. If the initiating transaction aborts, they are discarded. When it commits, they are made available for execution (in an indeterminate order). The set of available asynchronous method calls will be executed concurrently, within the available resources and subject to strategies that prevent excessive transaction aborts.

2.2 Blocking an Asynchronous Method

An asynchronous method may contain any number of calls to the system-supplied method:

```
blockUntil(<predicate>);
```

From the programmer's perspective, the code of an asynchronous method executes to completion only if all the executed calls of `blockUntil` within the method have predicates that evaluate to true. `blockUntil`'s implementation does nothing if the predicate holds, but otherwise it aborts the current transaction and re-executes it later (at a time when it is likely to succeed). This behavior is like that of `retry` in some systems (Harris et al. 2005).

2.3 Fragmenting an Asynchronous Method

A purely event-based model produces program structure that can be unpleasant and unstable. For example, if a previously non-blocking method call is modified to require a blocking action (e.g., a hash table is modified to use disk storage instead of main memory), the event-based style would require that the method, and all of its callers, gets split into two separate methods (a request and a response handler). This splitting is sometimes referred to as "stack ripping" (Adya et al. 2002). AME's solution is to allow an asynchronous method call to contain one or more invocations of the system method `yield()`. A `yield` call breaks a method into multiple atomic fragments. Importantly, these atomic fragments are delimited dynamically by the calls of `yield`, not statically scoped like explicit atomic blocks. With this enhancement, the overall execution of a program is guaranteed to be a serialization of its atomic fragments. We implement `yield` by committing the current transaction and starting a new one. A `blockUntil` call blocks execution of only the current atomic fragment (the code that follows the most recent `yield`), or equivalently, it retries only the transaction begun after the most recent `yield`.

2.4 External Side Effects

Actions with external side effects, such as I/O, are performed by asynchronous calls to an I/O library interface. The actual low-level I/O operations take place outside of transactions, either inside the AME runtime or in explicitly unprotected code. In order to support this and other access to legacy non-transacted code, we allow the following form:

```
unprotected { ... }
```

The unprotected code must use existing mechanisms for synchronization. The current atomic fragment ends before the unprotected statement, and a new one starts after it.

3. The AME Calculus

In our formal study, we focus on a small but expressive language. The language includes constructs for AME, as discussed above; it also includes higher-order functions and imperative features. We call it the AME calculus, though undoubtedly other calculi with AME are possible.

The syntax of the AME calculus is defined in Figure 1. This syntax is untyped; we introduce a type system in Section 6.2. We also give several formal semantics below. The syntax introduces syntactic categories of values, constants, variables, and expressions. The values are constants, variables, and lambda abstractions ($\lambda x.\, e$). In addition to values and to expressions of the forms `async` e, `blockUntil` e, and `unprotected` e, the expressions include notations for function application ($e f$), allocation (`ref` e, which allocates a new reference location and returns it after initializing it to

$$
\begin{array}{rcrcl}
V & \in & \mathit{Value} & = & c \mid x \mid \lambda x.\, e \\
c & \in & \mathit{Const} & = & \texttt{unit} \mid \texttt{false} \mid \texttt{true} \\
x, y & \in & \mathit{Var} & & \\
e, f & \in & \mathit{Exp} & = & V \\
 & & & \mid & e\ f \\
 & & & \mid & \texttt{ref}\ e \mid\ !e \mid e := f \\
 & & & \mid & \texttt{async}\ e \\
 & & & \mid & \texttt{blockUntil}\ e \\
 & & & \mid & \texttt{unprotected}\ e
\end{array}
$$

Figure 1. Syntax of the AME calculus.

the value of e), dereferencing ($!e$, which returns the contents in the reference location that is the value of e), and assignment ($e := f$, which sets the reference location that is the value of e to the value of f).

The syntax allows arbitrary nestings of `async`, `unprotected`, and `blockUntil`, and also allows `async` anywhere, not necessarily attached to a function call. In unprotected contexts, `blockUntil` e will behave roughly like "wait until e"—the precise meaning of this is defined by the semantics of Section 4. Practical embodiments of AME need not be as liberal in these respects.

As usual there is no difficulty in including other constructs. Several are definable:

- We abbreviate $(\lambda x.\ e')\ e$ to `let` $x = e$ `in` e'. We also abbreviate `let` $x = e$ `in` e' to $e; e'$ when x does not occur free in e'.
- We treat `yield` as syntactic sugar for `unprotected unit`.
- We can express "abort and retry" as `blockUntil false`.
- Traditional atomic blocks typically occur in the context of unprotected expressions, and differ from asynchronous calls in that they are supposed to be executed immediately, not in some indefinite future. We can express `atomic` e as:

  `let` $x =$ `ref false in`
  `async` $(e;$ `unprotected` $(x :=$ `true`$))$;
  `blockUntil` $!x$

  where x is a fresh variable that serves for signaling e's termination. The use of `unprotected` $(x :=$ `true`$)$ rather than simply $(x :=$ `true`$)$ ensures that, when this encoding is used in unprotected contexts (as intended), all accesses to x are done in unprotected contexts, thus conforming to the type system of Section 6.2.

4. Strong Semantics

This section defines a semantics for the AME calculus, intended to be a simple model of the constructs's expected behavior rather than of possible underlying implementation techniques. To this end, the semantics provides strong atomicity between the execution of transacted and non-transacted code, and it does not model roll-back, optimistic concurrency, and other low-level features. In Sections 7 and 8 we consider richer and weaker semantics that add these features.

4.1 States

As described in Figure 2, a state $\langle \sigma, T, e \rangle$ consists of the following components:

- a reference store σ,
- a collection of expressions T, which we call the pool,
- a distinguished active expression e.

$$
\begin{array}{rcrcl}
S & \in & \mathit{State} & = & \mathit{RefStore} \times \mathit{ExpSeq} \times \mathit{Exp} \\
\sigma & \in & \mathit{RefStore} & = & \mathit{RefLoc} \rightharpoonup \mathit{Value} \\
r & \in & \mathit{RefLoc} & \subset & \mathit{Var} \\
T & \in & \mathit{ExpSeq} & = & \mathit{Exp}^*
\end{array}
$$

Figure 2. State space.

A reference store σ is a finite mapping of reference locations to values. Reference locations are simply special kinds of variables that can be bound only by a reference store. We write *RefLoc* for the set of reference locations. We assume that *RefLoc* is infinite, so $\mathit{RefLoc} - \mathit{dom}(\sigma)$ is never empty. For every state $\langle \sigma, T, e \rangle$, we require that if $r \in \mathit{RefLoc}$ occurs free in $\sigma(r')$, in T, or in e, then $r \in \mathit{dom}(\sigma)$. This condition will be assumed for initial states and will be preserved by computation steps.

Informally, we may imagine that a computer includes a single special processor for performing "protected" work, occupied by the active expression, and an unbounded set of additional processors capable of doing "unprotected" work, dedicated to the pool. (This informal model is somewhat independent of the details of the AME calculus; indeed, we find it valuable in our work in the context of richer languages.) If no "unprotected" work is available, then expressions in the pool are simply waiting for the special processor. We identify expressions with threads of computation; the semantics does not describe stacks or other thread-specific data.

4.2 Steps

The evaluation of a program starts in an initial state $\langle \sigma, e, \texttt{unit} \rangle$ with a single expression in the pool and with `unit` as the distinguished active expression.

Evaluation then takes place according to rules (given below) that specify the behavior of the various constructs in the language. The execution of threads is interleaved in a non-deterministic manner, subject to atomicity constraints. Each evaluation step produces a new state. Given a state, the next state is determined by the next possible operation in the active expression or in one of the expressions in the pool. We model strong atomicity by allowing expressions in the pool to take steps only when the active expression is `unit`, thus preventing the interleaving of steps of unprotected and protected work.

In all cases, the next possible operation in an expression is found by decomposing the expression into an evaluation context and a subexpression that describes this operation. As usual, a context is an expression with a hole [ ], and an evaluation context is a context of a particular kind. Given a context $\mathcal{C}$ and an expression e, we write $\mathcal{C}[\ e\ ]$ for the result of placing e in the hole in $\mathcal{C}$. We use several kinds of evaluation contexts, defined in Figure 3:

- $\mathcal{P}$ evaluation contexts are for the execution of protected fragments: the position for evaluation is not under `unprotected`.
- $\mathcal{U}$ evaluation contexts are for the execution of unprotected fragments: the position for evaluation is under `unprotected`.
- $\mathcal{E}$ evaluation contexts allow us to manipulate `unprotected` values in the execution of unprotected fragments.

We also let some evaluation contexts be sequences of expressions with a hole:

- $\mathcal{F}$ evaluation contexts are of the form $T.\mathcal{U}.T'$, `unit` or of the form $T, \mathcal{P}$.

Thus, $\mathcal{F}[\ e\ ]$ is either of the form $T.\mathcal{U}[\ e\ ].T'$, `unit` or of the form $T, \mathcal{P}[\ e\ ]$. We write $e_0.\mathcal{F}[\ e_1\ ]$ as an abbreviation for $e_0.T.\mathcal{U}[\ e_1\ ].T'$, `unit` or $e_0.T, \mathcal{P}[\ e_1\ ]$, respectively.

$$\begin{array}{rcl}
\mathcal{P} & = & [\,] \mid \mathcal{P}\,e \mid V\,\mathcal{P} \mid \texttt{ref}\,\mathcal{P} \mid !\mathcal{P} \mid \mathcal{P} := e \mid r := \mathcal{P} \mid \texttt{blockUntil}\,\mathcal{P} \\
\mathcal{U} & = & \texttt{unprotected}\,\mathcal{E} \mid \mathcal{U}\,e \mid V\,\mathcal{U} \mid \texttt{ref}\,\mathcal{U} \mid !\mathcal{U} \mid \mathcal{U} := e \mid r := \mathcal{U} \mid \texttt{blockUntil}\,\mathcal{U} \\
\mathcal{E} & = & [\,] \mid \mathcal{E}\,e \mid V\,\mathcal{E} \mid \texttt{ref}\,\mathcal{E} \mid !\mathcal{E} \mid \mathcal{E} := e \mid r := \mathcal{E} \mid \texttt{blockUntil}\,\mathcal{E} \mid \texttt{unprotected}\,\mathcal{E} \\
\mathcal{F} & = & T.\mathcal{U}.T', \texttt{unit} \mid T, \mathcal{P}
\end{array}$$

Figure 3. Evaluation contexts.

$$\begin{array}{llll}
\langle \sigma, \mathcal{F}[\,(\lambda x.\,e)\,V\,]\rangle & \longmapsto_s & \langle \sigma, \mathcal{F}[\,e[V/x]\,]\rangle & \text{(Trans Appl)}_s \\
\langle \sigma, \mathcal{F}[\,\texttt{ref}\,V\,]\rangle & \longmapsto_s & \langle \sigma[r \mapsto V], \mathcal{F}[\,r\,]\rangle & \text{(Trans Ref)}_s \\
 & & \text{if } r \in \mathit{RefLoc} - \mathit{dom}(\sigma) & \\
\langle \sigma, \mathcal{F}[\,!r\,]\rangle & \longmapsto_s & \langle \sigma, \mathcal{F}[\,V\,]\rangle & \text{(Trans Deref)}_s \\
 & & \text{if } \sigma(r) = V & \\
\langle \sigma, \mathcal{F}[\,r := V\,]\rangle & \longmapsto_s & \langle \sigma[r \mapsto V], \mathcal{F}[\,\texttt{unit}\,]\rangle & \text{(Trans Set)}_s \\
\langle \sigma, \mathcal{F}[\,\texttt{async}\,e\,]\rangle & \longmapsto_s & \langle \sigma, e.\mathcal{F}[\,\texttt{unit}\,]\rangle & \text{(Trans Async)}_s \\
\langle \sigma, \mathcal{F}[\,\texttt{blockUntil true}\,]\rangle & \longmapsto_s & \langle \sigma, \mathcal{F}[\,\texttt{unit}\,]\rangle & \text{(Trans Block)}_s \\
\langle \sigma, T, \mathcal{P}[\,\texttt{unprotected}\,e\,]\rangle & \longmapsto_s & \langle \sigma, T.\mathcal{P}[\,\texttt{unprotected}\,e\,], \texttt{unit}\rangle & \text{(Trans Unprotect)}_s \\
\langle \sigma, T.\mathcal{E}[\,\texttt{unprotected}\,V\,].T', \texttt{unit}\rangle & \longmapsto_s & \langle \sigma, T.\mathcal{E}[\,V\,].T', \texttt{unit}\rangle & \text{(Trans Close)}_s \\
\langle \sigma, T.e.T', \texttt{unit}\rangle & \longmapsto_s & \langle \sigma, T.T', e\rangle & \text{(Trans Activate)}_s
\end{array}$$

Figure 4. Transition rules of the abstract machine (strong).

Figure 4 gives rules that specify the transition relation that takes execution from one state to the next. The string "Trans" in the names of the rules refers to "transition" rules, not to "transaction". In these rules, we write $e[V/x]$ for the result of the capture-free substitution of V for x in e, and write $\sigma[r \mapsto V]$ for the store that agrees with σ except at r, which is mapped to V. The subscript s in $\longmapsto_s$ indicates that this is a strong semantics.

Were `yield` not syntactic sugar we could have the two extra rules:

$$\begin{array}{lll}
\langle \sigma, T, \mathcal{P}[\,\texttt{yield}\,]\rangle & \longmapsto_s & \langle \sigma, T.\mathcal{P}[\,\texttt{yield}\,], \texttt{unit}\rangle \\
\langle \sigma, T.\mathcal{E}[\,\texttt{yield}\,].T', e'\rangle & \longmapsto_s & \langle \sigma, T.\mathcal{E}[\,\texttt{unit}\,].T', e'\rangle
\end{array}$$

These rules are easily derived from those of Figure 4 and the definition of `yield` as `unprotected unit`.

5. Problems with Weak Atomicity

The strong semantics of Section 4 is intended to reflect a programmer's intuition about the behavior of the AME constructs, but it is unlikely to be practical to implement in software without language restrictions. In particular, the main purpose of using unprotected regions is to interact with the operating system and other legacy code that cannot easily be changed; implementations that offer strong atomicity by recompiling unprotected code do not support this purpose.

In this section we discuss the ways in which different implementations of STM can give behavior that differs from the strong semantics. For the purposes of this discussion, we write examples informally (rather than in a calculus like that of Section 3) for convenience, and in order to emphasize the relevance of these examples to practical code. However, we use the strong semantics as a point of reference.

5.1 Review of Examples from Shpeisman et al.

The first set of examples, in Figure 5, comes from work on implementing strong atomicity (Shpeisman et al. 2007). In all cases code is protected (i.e., runs transactionally) unless it is contained in an `unprotected` block. Shpeisman et al. discuss how these examples can cause unexpected behavior with existing STM implementations—in particular, almost all of these problems occur with Bartok-STM because of its use of in-place update and lazy conflict detection. Bartok-STM does not exhibit the GIR and GLU problems; these occur in other STMs (e.g., (Harris and Fraser 2003)) which can buffer data at a coarser granularity than individual fields: a transaction committing or being rolled back can involve writes that spill over onto adjacent locations.

5.2 Are These Problems Data Races?

One may reasonably ask "Do these problems matter?" because most of the examples in Figure 5 intuitively have data races. For instance, in the Non-Repeatable Reads (NR) problem, there is no synchronization between the non-transacted store to `x` and the transacted read from `x`. In fact, almost all the examples in the figure involve two threads accessing `x` without any synchronization between them. (The sole exception is GLU and this problem is readily solved by making the STM buffer data on a per-field basis. Bartok-STM already does this.)

Unfortunately, although these examples could be considered to have data races, other examples *are* free from data races at the source level (both intuitively and with respect to formal definitions below) but *do not* obey the strong semantics with many STM systems. For concreteness, again we focus on how these problems can occur with Bartok-STM; however, we believe that variants of the problems of Sections 5.4–5.5 affect all extant STM systems that allow data to be shared between protected and unprotected code.

(NR) Non-repeatable reads: `r1!=r2`

```
r1 = x;                     unprotected {
r2 = x;                       x = 1;
                            }
```

(ILU) Intermediate lost updates: `x==1`

```
r1 = x;                     unprotected {
x = r1 + 1;                   x = 10;
                            }
```

(IDR) Intermediate dirty reads: `r1==1`

```
x ++;                       unprotected {
x ++;                         r1 = x;
                            }
```

(SLU) Speculative lost updates: `x==0`

```
if (y==0) {                 unprotected {
   x = 1;                     x = 2;
   // Abort                   y = 1;
}                           }
```

(SDR) Speculative dirty reads: `x==0, y==1`

```
if (y == 0) {               unprotected {
  x = 1;                      if (x == 1) {
  // Abort                      y = 1;
}                           } }
```

(OW) Overlapped writes: `r1==0`

```
o1.val = 1;                 unprotected {
x = o1;                       r1 = -1;
                              if (x != null) {
                                r1 = x.val;
                            } }
```

(BW) Buffered writes: `r2!=r3` *or* `r1.val!=0`

```
// Initially x!=null, x.val==1
r1 = x;                     if (x != null) {
x = null;                     x.val++;
unprotected {               }
  r2 = r1.val;
  r3 = r1.val;
  r1.val = 0;
}
```

(GIR) Granular inconsistent reads: `r==0`

```
r = -1;                     unprotected {
atomic {                      y.g = 1;
  y.f = ...;                  x = 1;
  if (x==1) {               }
    r = y.g;
} }
```

(GLU) Granular lost update: `x.g==0`

```
x.f = 1;                    unprotected {
                              x.g = 1;
                            }
```

Figure 5. Example problems from Shpeisman et al. (2007). Unless otherwise noted, all fields initially hold 0. Registers `r`, `r1`, `r2`, and `r3` are thread-local.

5.3 Zombie Transactions

The first example concerns *zombie transactions* that access more data than would be touched in any serialization. Consider the following two atomic actions `A1` and `A2` that run concurrently with the unprotected block `U1`:

```
// A1                 // A2        // U1
r1 = u;               u++;         unprotected {
r2 = v;               v++;           r1 = x;
if (r1 != r2) {                    }
  x = 42;
}
```

Informally, one may reason that both serialization orders for `A1` and `A2` will maintain the invariant `u==v`, so the condition `r1!=r2` should never be satisfied, `A1` will never write to `x`, and therefore there is no data race with `U1`'s read from `x`.

However, with Bartok-STM, `A2` may run in its entirety in between `A1`'s reads from `u` and `v`, causing `A1` to write to `x` before the conflict is detected. Despite the conflict detection and any resulting roll-back, `U1` may see this write. This kind of example is particularly problematic in native code. For instance, suppose that instead of writing to `x`, `A1` indexes an array `x[r1-r2]`: in a language without bounds checking, it may actually write to any location dependent on the number of increments performed in `A2`.

5.4 Privatization

A second example is the *privatization problem* in which a piece of data is sometimes accessed from protected code and sometimes accessed directly. Consider these code fragments, with one thread running `A1` and then `U1`, and a second thread running `A2`:

```
// Initially: x_shared=true, x=0

// A1                       // A2
x_shared = false;           if (x_shared) {
// U1                         x = 42;
unprotected {               }
  x ++;
}
```

Informally, one may reason that this code has no data races: `x_shared` is always accessed transactionally and, by the time `U1` accesses `x` non-transactionally, `A1` has already been executed and either `A2` is serialized before `A1` (so the accesses to `x` cannot race) or `A2` is serialized after `A1` (so it will see that `x_shared` is `false`).

With Bartok-STM, it is possible for `A1` to execute in its entirety between `A2`'s read from `x_shared` and its write to `x` and then for `U1`'s accesses to `x` to race with `A2`. In Bartok-STM the problem is therefore similar to that of Section 5.3 in that it occurs because `A2` continues to execute as a zombie. However, the same problem can occur without zombies on STMs that buffer transactional updates and write them back on commit (Harris and Fraser 2003): `A2`'s write-back to `x` may race with `U1`'s accesses.

5.5 Publication

A final example is the *publication problem* in which a piece of data is initially thread-private and then becomes shared:

```
// Initially: x_shared=false, x=0

// U1                        // A2
unprotected {                r1 = -1;
  x = 42;                    if (x_shared) {
}                              r1 = x;
// A1                        }
x_shared = true;
```

Once again, one may reason informally that this code has no data races: `x_shared` is always accessed transactionally and, when it is set by `A1`, the update to `x` has already been performed. If `A1` is serialized before `A2` then `A2` will see both updates.

The problem here is more subtle and relates to more of the language than just the STM implementation: there is no indication in the source code that the ordering between `A2`'s reads from `x` and `x_shared` is important. If they are re-ordered during compilation then the implementation of `A2` may read from `x` *before* `U1`, and then read from `x_shared` *after* `A1`, leaving `A2` serialized after `A1`, but with `r1==0`. A similar lock-based program, placing `A1` and `A2` in regions protected by the same lock, is correctly synchronized under the Java memory model (Manson et al. 2005b). As with our strong semantics, it would give either `r1==-1` or `r1==42`.

6. Violation-freedom and Separation

In Section 5, we show example programs that are not executed correctly by STM systems. In some cases, these are programs with data races, while in others the problems arise because (despite the absence of apparent data races) a variable `x` is accessed from both protected and unprotected code in the implementations.

In this section we present two criteria that formalize the separation of protected and unprotected code, in the AME calculus. The first criterion, violation-freedom, says that, dynamically, data cannot be accessed with and without protection at the same time. This criterion allows us to say, formally, that the examples of Sections 5.3–5.5 are correctly synchronized, while most of the examples of Section 5.1 are not. The second criterion, separation, is embodied in a static discipline that guarantees that protected and unprotected computations do not use the same reference locations. As we prove, separation implies violation-freedom. In Sections 7 and 8, we show that, by restricting ourselves to programs that meet these criteria, we can enable the use of efficient and correct lower-level semantics.

6.1 Violation-free Executions

We define a condition according to which data cannot be accessed with and without protection at the same time in different threads.

Given a state $\langle \sigma, e_1. \cdots .e_n, e\rangle$, there is a violation on a location r if $e_i = \mathcal{U}[\ f\ ]$ for some $i = 1..n$ and $e = \mathcal{P}[\ f'\ ]$ where f and f' are reads or writes on r (that is, expressions $!r$ or $r := \ldots$), and at least one of them is a write ($r := \ldots$). A computation is violation-free if none of its states have violations for any locations. (Analogously, we could define races, in which we would also consider conflicts within $e_1. \cdots .e_n$; every violation is a race but not every race is a violation.)

A possible programming discipline is to require that programs never generate violations in the strong semantics. Under this discipline, a state $\langle \sigma, T, e\rangle$ is good if all strong computations that start from this state are violation-free. The use of the strong semantics is significant: programmers should not have to understand lower-level implementations. However, analogous criteria apply to lower-level implementations, and might be of benefit in compiler optimizations. Some of our lemmas say that the absence of violations in the strong semantics implies the absence of violations in certain lower-level implementations.

6.2 Separation

The type system described in this section embodies a discipline in which protected and unprotected computations do not use the same portions of the reference store. They may however communicate via variables.

The type system is defined in Figure 6, using judgments and rules for reasoning about the judgments. The core of the type system is the set of rules for the judgment $E\,;p \vdash e : t$ (read "e is a well-typed expression of type t in typing environment E with effect p"). The intent is that, if this judgment holds, then e yields values of type t with effect p, and the free variables of e are given bindings consistent with the typing environment E. When p is `P`, this means that the evaluation of e accesses only the part of the reference store for protected computations; when p is `U`, this means that the evaluation of e accesses only the rest of the store. The typing environment E is organized as a sequence of bindings, and we use $\emptyset$ to denote the empty environment. Similarly, $s \to^p t$ is the type of function that take arguments of type s and yield results of type t with effect p.

The type system introduces a sharp distinction between "`P` code" and "`U` code". The type system is thus deliberately simple; various elaborations are possible, mostly along standard lines, but we do not need them for our present purposes.

Judgments

$E \vdash \diamond$	E is a well-formed typing environment
$E\,;p \vdash e : t$	e is a well-typed expression of type t in E with effect p

Rules

$$\emptyset \vdash \diamond \qquad \text{(Env } \emptyset)$$

$$\frac{E \vdash \diamond \quad x \notin dom(E)}{E, x : t \vdash \diamond} \qquad \text{(Env } x)$$

$$\frac{E \vdash \diamond}{E\,;p \vdash \texttt{unit} : \texttt{Unit}} \qquad \text{(Exp Unit)}$$

$$\frac{E \vdash \diamond}{E\,;p \vdash \texttt{false} : \texttt{Bool}} \qquad \text{(Exp Bool } \texttt{false})$$

$$\frac{E \vdash \diamond}{E\,;p \vdash \texttt{true} : \texttt{Bool}} \qquad \text{(Exp Bool } \texttt{true})$$

$$\frac{E, x : t, E' \vdash \diamond}{E, x : t, E'\,;p \vdash x : t} \qquad \text{(Exp } x)$$

$$\frac{E, x : s\,;p \vdash e : t}{E\,;q \vdash \lambda x.\, e : s \to^p t} \qquad \text{(Exp Fun)}$$

$$\frac{E\,;p \vdash e_1 : s \to^p t \quad E\,;p \vdash e_2 : s}{E\,;p \vdash e_1\ e_2 : t} \qquad \text{(Exp Appl)}$$

$$\frac{E\,;p \vdash e : t}{E\,;p \vdash \texttt{ref}\ e : \texttt{Ref}_p\ t} \qquad \text{(Exp Ref)}$$

$$\frac{E\,;p \vdash e : \texttt{Ref}_p\ t}{E\,;p \vdash !e : t} \qquad \text{(Exp Deref)}$$

$$\frac{E\,;p \vdash e_1 : \texttt{Ref}_p\ t \quad E\,;p \vdash e_2 : t}{E\,;p \vdash e_1 := e_2 : \texttt{Unit}} \qquad \text{(Exp Set)}$$

$$\frac{E\,;\texttt{P} \vdash e : \texttt{Unit}}{E\,;q \vdash \texttt{async}\ e : \texttt{Unit}} \qquad \text{(Exp Async)}$$

$$\frac{E\,;p \vdash e : \texttt{Bool}}{E\,;p \vdash \texttt{blockUntil}\ e : \texttt{Unit}} \qquad \text{(Exp Block)}$$

$$\frac{E\,;\texttt{U} \vdash e : t}{E\,;p \vdash \texttt{unprotected}\ e : t} \qquad \text{(Exp Unprotect)}$$

Figure 6. The first-order type system for separation.

The following small example illustrates the restrictions that the type system imposes:

```
let x = ref V in
let y = ref true in
async (y := false;
       unprotected z := !x);
async (blockUntil !y;
       x := V')
```

where V and V' are distinct values. Intuitively, the contents of the reference location y indicates whether x is shared; setting that location to `false` amounts to a privatization. This program is not permitted by the type system, because the reference location that is the value of x is used in both protected and unprotected computations.

On the other hand, the following variant of the program is permitted by the type system:

$$\begin{array}{l}
\texttt{let}\ x = \texttt{ref}\ V\ \texttt{in} \\
\texttt{let}\ y = \texttt{ref}\ \texttt{true}\ \texttt{in} \\
\texttt{async}\ (y := \texttt{false}; \\
\qquad \texttt{let}\ x' = !x\ \texttt{in}\ (\texttt{unprotected}\ z := x')); \\
\texttt{async}\ (\texttt{blockUntil}\ !y; \\
\qquad x := V')
\end{array}$$

Here, the reference location in question is used only in protected computations; its value is put into a local variable x' for use in an unprotected computation in the same thread.

In order to prove the soundness of the type system, we extend it to states $\langle \sigma, T, e \rangle$. We write

$$E \vdash \langle \sigma, e_1. \cdots .e_n, e \rangle$$

if

- $dom(\sigma) = dom(E) \cap RefLoc$,
- for all $r \in dom(\sigma)$, there exist t and p such that $E(r) = \texttt{Ref}_p\ t$ and $E\,; p \vdash \sigma(r) : t$,
- $E\,; \texttt{P} \vdash e_i : \texttt{Unit}$ for all $i = 1..n$,
- $E\,; \texttt{P} \vdash e : \texttt{Unit}$.

We say that $\langle \sigma, e_1. \cdots .e_n, e \rangle$ is well-typed if there exist E such that $E \vdash \langle \sigma, e_1. \cdots .e_n, e \rangle$. We write $\longmapsto_s^*$ for the reflexive-transitive closure of $\longmapsto_s$. We obtain that typability is preserved by computation (that is, by $\longmapsto_s^*$):

THEOREM 6.1 (Preservation of Typability). *If $\langle \sigma, T, e \rangle$ is well-typed and $\langle \sigma, T, e \rangle \longmapsto_s^* \langle \sigma', T', e' \rangle$, then $\langle \sigma', T', e' \rangle$ is well-typed.*

This theorem helps in relating the type system to the absence of violations, and it serves as the basis for analogous results for lower-level semantics, below.

We also obtain a progress result, which characterizes when a computation may stop and implies that computations do not get stuck in unexpected ways (for instance, by applying a boolean as though it were a function). This progress result is partly a sanity check; stronger ones are viable.

THEOREM 6.2 (Progress). *If $\langle \sigma, T, e \rangle$ is well-typed, the only free variables in $\langle \sigma, T, e \rangle$ are reference locations, and $\langle \sigma, T, e \rangle \longmapsto_s^* \langle \sigma', T', e' \rangle$, then:*

1. *e' is* `unit` *and T' is empty; or*
2. *e' is of the form $\mathcal{P}[\ \texttt{blockUntil false}\ ]$; or*
3. *$\langle \sigma', T', e' \rangle \longmapsto_s \langle \sigma'', T'', e'' \rangle$ for some $\langle \sigma'', T'', e'' \rangle$.*

6.3 Comparing Separation with Violation-freedom

Violation-freedom is a clear but undecidable dynamic criterion. The type system for separation provides a sufficient condition for violation-freedom. As a corollary to Theorem 6.1, we obtain:

COROLLARY 6.3. *If $\langle \sigma, T, e \rangle$ is well-typed, then all strong computations that start from $\langle \sigma, T, e \rangle$ are violation-free.*

As suggested above, separation appears to be more robust than violation-freedom (for instance, less fragile in the presence of compiler optimizations).

7. Weak Semantics with Roll-back

Having introduced the violation-freedom and separation criteria in Section 6, we can examine their impact on the use of weaker semantics that model some of the implementation techniques used by actual STMs: if a program meets one or other of the criteria, then which implementation techniques can be used while respecting the strong semantics?

In this section we define a semantics that models weak atomicity, allowing steps of unprotected code to be interleaved with steps of protected code, and also models eager versioning, in which transactions make in-place updates to the heap and are rolled back if they abort for some reason. This semantics still serializes transactions: only one piece of protected code can run at a time. We show that this weak semantics is correct for violation-free programs. Even without concurrency between transactions, this weak semantics is still interesting from a practical point of view as well as a theoretical one—for instance to provide roll-back on a uni-processor real-time system (Manson et al. 2005a). We consider concurrency between transactions in Section 8.

$$\begin{array}{rcrcl}
S & \in & \mathit{State} & = & \mathit{RefStore} \times \mathit{ExpSeq} \times \\
 & & & & \mathit{Exp} \times \mathit{Exp} \times \mathit{Log} \times \mathit{ExpSeq} \\
\sigma & \in & \mathit{RefStore} & = & \mathit{RefLoc} \rightharpoonup \mathit{Value} \\
l & \in & \mathit{Log} & = & (\mathit{RefLoc} \times \mathit{Value})^* \\
r & \in & \mathit{RefLoc} & \subset & \mathit{Var} \\
T, P & \in & \mathit{ExpSeq} & = & \mathit{Exp}^*
\end{array}$$

Figure 7. State space, with roll-back.

7.1 States

Figure 7 defines states for the semantics with roll-back. A state $\langle \sigma, T, e, f, l, P \rangle$ consists of the following components:

- σ, T, and e, which are as usual,
- f, an expression that, through computation, has yielded e (and which we call the origin of e),
- l, a list of memory locations and their values, to be used as a log in undos,
- P, a list of "pending" threads to be forked upon commit.

Much as in Section 4.1, for every state $\langle \sigma, T, e, f, l, P \rangle$, we require that if $r \in RefLoc$ occurs free in $\sigma(r')$, in T, in e, in f, in l, or in P, then $r \in dom(\sigma)$. This condition will be assumed for initial states and will be preserved by computation steps.

We write each pair in l in the form $[r \mapsto V]$, we let $dom(l)$ be the set of locations r for which l is defined, and when $r \in dom(l)$ we write $l(r)$ for the value V to which r is mapped.

7.2 Steps

Figure 8 gives the rules of this semantics. The intent is that, upon a roll-back caused by e, the origin expression f is added back to T and the undos described in l are performed. The semantics has a few subtleties.

- As in some practical STM implementations (Saha et al. 2006; Harris et al. 2006), the undos described in l are performed individually rather than as one atomic step. We pick an arbitrary order.
- Allocations are not undone. If they were, we could cause dangling pointers in programs with race conditions—and we believe that dangling pointers should be avoided even in programs with synchronization errors. Again, this detail is inspired by practical STMs (Harris et al. 2005).
- No undo facilities are provided for unprotected computations.
- Since this is a weak semantics, unprotected computations may be interleaved with protected computations, and even with the roll-backs of protected computations.

$\langle \sigma, T, \mathcal{P}[\ (\lambda x.\ e)\ V\ ], f, l, P \rangle$	$\longmapsto_{rw}$	$\langle \sigma, T, \mathcal{P}[\ e[V/x]\ ], f, l, P \rangle$	(Trans Appl P)$_{rw}$
$\langle \sigma, T.\mathcal{U}[\ (\lambda x.\ e)\ V\ ].T', e', f, l, P \rangle$	$\longmapsto_{rw}$	$\langle \sigma, T.\mathcal{U}[\ e[V/x]\ ].T', e', f, l, P \rangle$	(Trans Appl U)$_{rw}$
$\langle \sigma, T, \mathcal{P}[\ \texttt{ref}\ V\ ], f, l, P \rangle$	$\longmapsto_{rw}$	$\langle \sigma[r \mapsto V], T, \mathcal{P}[\ r\ ], f, l, P \rangle$ if $r \in \mathit{RefLoc} - \mathit{dom}(\sigma)$	(Trans Ref P)$_{rw}$
$\langle \sigma, T.\mathcal{U}[\ \texttt{ref}\ V\ ].T', e, f, l, P \rangle$	$\longmapsto_{rw}$	$\langle \sigma[r \mapsto V], T.\mathcal{U}[\ r\ ].T', e, f, l, P \rangle$ if $r \in \mathit{RefLoc} - \mathit{dom}(\sigma)$	(Trans Ref U)$_{rw}$
$\langle \sigma, T, \mathcal{P}[\ !r\ ], f, l, P \rangle$	$\longmapsto_{rw}$	$\langle \sigma, T, \mathcal{P}[\ V\ ], f, l, P \rangle$ if $\sigma(r) = V$	(Trans Deref P)$_{rw}$
$\langle \sigma, T.\mathcal{U}[\ !r\ ].T', e, f, l, P \rangle$	$\longmapsto_{rw}$	$\langle \sigma, T.\mathcal{U}[\ V\ ].T', e, f, l, P \rangle$ if $\sigma(r) = V$	(Trans Deref U)$_{rw}$
$\langle \sigma, T, \mathcal{P}[\ r := V\ ], f, l, P \rangle$	$\longmapsto_{rw}$	$\langle \sigma[r \mapsto V], T, \mathcal{P}[\ \texttt{unit}\ ], f, l', P \rangle$ where $l' =$ if $r \in \mathit{dom}(l)$ then l else $l.[r \mapsto \sigma(r)]$	(Trans Set P)$_{rw}$
$\langle \sigma, T.\mathcal{U}[\ r := V\ ].T', e, f, l, P \rangle$	$\longmapsto_{rw}$	$\langle \sigma[r \mapsto V], T.\mathcal{U}[\ \texttt{unit}\ ].T', e, f, l, P \rangle$	(Trans Set U)$_{rw}$
$\langle \sigma, T, \mathcal{P}[\ \texttt{async}\ e\ ], f, l, P \rangle$	$\longmapsto_{rw}$	$\langle \sigma, T, \mathcal{P}[\ \texttt{unit}\ ], f, l, e.P \rangle$	(Trans Async P)$_{rw}$
$\langle \sigma, T.\mathcal{U}[\ \texttt{async}\ e\ ].T', e', f, l, P \rangle$	$\longmapsto_{rw}$	$\langle \sigma, e.T.\mathcal{U}[\ \texttt{unit}\ ].T', e', f, l, P \rangle$	(Trans Async U)$_{rw}$
$\langle \sigma, T, \mathcal{P}[\ \texttt{blockUntil true}\ ], f, l, P \rangle$	$\longmapsto_{rw}$	$\langle \sigma, T, \mathcal{P}[\ \texttt{unit}\ ], f, l, P \rangle$	(Trans Block `true` P)$_{rw}$
$\langle \sigma, T.\mathcal{U}[\ \texttt{blockUntil true}\ ].T', e, f, l, P \rangle$	$\longmapsto_{rw}$	$\langle \sigma, T.\mathcal{U}[\ \texttt{unit}\ ].T', e, f, l, P \rangle$	(Trans Block `true` U)$_{rw}$
$\langle \sigma, T, \mathcal{P}[\ \texttt{blockUntil false}\ ], f, \emptyset, P \rangle$	$\longmapsto_{rw}$	$\langle \sigma, f.T, \texttt{unit}, \texttt{unit}, \emptyset, \emptyset \rangle$	(Trans Block `false` Restore)$_{rw}$
$\langle \sigma, T, \mathcal{P}[\ \texttt{blockUntil false}\ ], f, l.[r \mapsto V], P \rangle$	$\longmapsto_{rw}$	$\langle \sigma[r \mapsto V], T, \mathcal{P}[\ \texttt{blockUntil false}\ ], f, l, P \rangle$	(Trans Block `false` Undo)$_{rw}$
$\langle \sigma, T, \mathcal{P}[\ \texttt{unprotected}\ e\ ], f, l, P \rangle$	$\longmapsto_{rw}$	$\langle \sigma, T.P.\mathcal{P}[\ \texttt{unprotected}\ e\ ], \texttt{unit}, \texttt{unit}, \emptyset, \emptyset \rangle$	(Trans Unprotect)$_{rw}$
$\langle \sigma, T, \texttt{unit}, f, l, P \rangle$	$\longmapsto_{rw}$	$\langle \sigma, T.P, \texttt{unit}, \texttt{unit}, \emptyset, \emptyset \rangle$	(Trans Done)$_{rw}$
$\langle \sigma, T.\mathcal{E}[\ \texttt{unprotected}\ V\ ].T', e, f, l, P \rangle$	$\longmapsto_{rw}$	$\langle \sigma, T.\mathcal{E}[\ V\ ].T', e, f, l, P \rangle$	(Trans Close)$_{rw}$
$\langle \sigma, T.e.T', \texttt{unit}, \texttt{unit}, \emptyset, \emptyset \rangle$	$\longmapsto_{rw}$	$\langle \sigma, T.T', e, e, \emptyset, \emptyset \rangle$	(Trans Activate)$_{rw}$

Figure 8. Transition rules of the abstract machine, with roll-back (weak).

- In the strong semantics of Section 4, there is no analogue for the list of pending threads P. Instead, the corresponding threads are put into T, but they cannot make immediate progress.

7.3 Correctness

The goal of this section is to establish the correctness of the weak semantics with roll-back as an implementation of the simpler strong semantics without roll-back, assuming the absence of violations. As the examples of Section 5 suggest, the violation-freedom hypothesis is needed. More specifically, we prove that an intermediate strong semantics with roll-back implements the strong semantics without roll-back; this result does not require violation-freedom. On the other hand, the weak semantics with roll-back is a correct implementation of the strong semantics with roll-back only under some assumptions. We obtain the following theorem:

THEOREM 7.1 (Correctness). *Assume that all strong computations that start from the state $\langle \sigma, T, \texttt{unit} \rangle$ are violation-free. Consider a weak computation with roll-back*

$$\langle \sigma, T, \texttt{unit}, \texttt{unit}, \emptyset, \emptyset \rangle \longmapsto_{rw}^{*} \langle \sigma', T', \texttt{unit}, \texttt{unit}, \emptyset, \emptyset \rangle$$

Then there is a strong computation

$$\langle \sigma, T, \texttt{unit} \rangle \longmapsto_{s}^{*} \langle \sigma'', T'', \texttt{unit} \rangle$$

for some σ'' and T'' such that σ' is an extension of σ'' and $T'' = T$ up to reordering.

This theorem is restricted to computations that lead to states of a particular form, in particular with an active expression `unit`. In general, when the active expression is not `unit`, the intermediate store σ' may be one that cannot be obtained by strong computations. Moreover, this theorem does not yield a strong computation with exactly the same final store: intuitively, the computation with roll-backs may allocate additional locations, and those are not deallocated. However, the two final stores coincide at all accessible locations: our invariant on states implies that both stores are defined (and equal) at all referenced locations.

We deduce a correctness theorem for well-typed programs:

COROLLARY 7.2. *Assume that $\langle \sigma, T, \texttt{unit} \rangle$ is well-typed. Consider a weak computation with roll-back*

$$\langle \sigma, T, \texttt{unit}, \texttt{unit}, \emptyset, \emptyset \rangle \longmapsto_{rw}^{*} \langle \sigma', T', \texttt{unit}, \texttt{unit}, \emptyset, \emptyset \rangle$$

Then there is a strong computation

$$\langle \sigma, T, \texttt{unit} \rangle \longmapsto_{s}^{*} \langle \sigma'', T'', \texttt{unit} \rangle$$

for some σ'' and T'' such that σ' is an extension of σ'' and $T'' = T$ up to reordering.

$$\begin{array}{rclcl}
S & \in & \mathit{State} & = & \mathit{RefStore} \times \mathit{ExpSeq} \times \mathit{TrySeq} \times \mathit{Log} \\
\sigma & \in & \mathit{RefStore} & = & \mathit{RefLoc} \rightharpoonup \mathit{Value} \\
l & \in & \mathit{Log} & = & (\mathit{RefLoc} \times \mathit{Value})^* \\
r & \in & \mathit{RefLoc} & \subset & \mathit{Var} \\
T, P & \in & \mathit{ExpSeq} & = & \mathit{Exp}^* \\
O & \in & \mathit{TrySeq} & = & \mathit{Try}^* \\
d & \in & \mathit{Try} & = & \mathit{Exp} \times \mathit{Exp} \times \mathit{Accesses} \times \mathit{ExpSeq} \\
a & \in & \mathit{Accesses} & = & \mathit{RefLoc}^*
\end{array}$$

Figure 9. State space, with optimistic concurrency.

8. Weak Semantics with Optimistic Concurrency

Building on the study of roll-back, we treat a difficult extension of the operational semantics in which multiple active expressions are evaluated simultaneously, with roll-backs in case of conflict. We ground our work on important aspects of Bartok-STM, as above, by making in-place updates to the reference store and using lazy conflict detection—although alternative strategies might be easier to analyze.

Like roll-back, optimistic concurrency raises correctness issues. Interestingly, and unlike for our semantics of Section 7, violation-freedom is *not* a sufficient condition for correctness in this case. As in our examples of Section 5.3–5.4, a program can be violation-free under the strong semantics but have lower-level violations because of zombie transactions. Nevertheless, we show that if a program is well-typed in the type system of Section 6.2 then its weak semantics is correct with respect to the strong semantics.

8.1 States

As described in Figure 9, states become more complex for this semantics. In addition to the components σ, T, and l that appear in the semantics with roll-back, here we have a list of tuples instead of a single active expressions e and its origin expression f. Each of the tuples is called a try, and consists of the following components:

- an active expression e,
- its origin expression f, as in the semantics with roll-back,
- a description of the accesses that e has performed, which are used for conflict detection and which here is simply a list of reference locations,
- a list P of threads to be forked upon commit.

Clearly these components could be refined further in more elaborate, realistic schemes. For instance, conflict detection could distinguish reads and writes, possibly with timestamps; moreover, the log used for undos could contain additional information in order to support more selective undos. (Actual STM implementations typically resolve conflicts by aborting some transactions and committing others.) We prefer to avoid this tedious book-keeping since it might obscure the presentation. Even in the present form, the semantics exhibits challenging features.

8.2 Steps

Figure 10 gives the rules of this semantics. They rely on the following definitions:

- (e_i, f_i, a_i, P_i) and (e_j, f_j, a_j, P_j) conflict if a_i and a_j have at least one element in common.
- (e, f, a, P) conflicts with O if (e, f, a, P) conflicts with some try in O.
- O conflicts if it contains two distinct tries that conflict.
- Given a log l and a list of reference locations a, $l - a$ is the log obtained from l by restricting to reference locations not in a.
- If O is $(e_1, f_1, a_1, P_1). \cdots .(e_n, f_n, a_n, P_n)$ then *origin*(O) is the list $f_1. \cdots .f_n$.
- σl is the result of applying all elements of l to σ.

The rules allow for conflicts to be detected as soon as they occur, but they do not require it. For simplicity, the rules do not include some secondary features sufficiently explored in the semantics with roll-back of Section 7. In particular, undos are atomic. Moreover, there is no special treatment for `blockUntil false`; the rules simply allow undo to happen at any point (possibly because of conflicts, but also possibly because of `blockUntil false`).

In this semantics, each transition has the form

$$\langle \sigma, T, O, l \rangle \longmapsto_{ow} \langle \sigma', T', O', l' \rangle$$

In many cases, a transition is defined in terms of a context that has a hole either in T and in T', or in O and in O'. We say that the transition is protected if the hole is in O and in O', and say that the transition is unprotected if the hole is in T and in T'. By definition, we have:

- transitions that are instances of (Trans . . . P)$_{ow}$ are always protected;
- transitions that are instances of (Trans . . . U)$_{ow}$ or of (Trans Close)$_{ow}$ are always unprotected;
- transitions that are instances of (Trans Undo)$_{ow}$, (Trans Unprotect)$_{ow}$, (Trans Done)$_{ow}$, or (Trans Activate)$_{ow}$ are neither protected nor unprotected.

8.3 Correctness

As explained above, the absence of high-level violations does not in general suffice for correctness. It is plausible that the absence of lower-level violations would suffice for correctness. This result could be adequate as a basis for compiler optimizations, but would not be fully satisfactory—programmers should not be aware of the details of this lower-level semantics. Instead, we rely on the type system for separation.

We do not modify the source typing rules of Section 6.2, but we do extend them to the states defined in this section. We write:

$$E \vdash \langle \sigma, T, O, l \rangle$$

if

- $\mathit{dom}(\sigma) = \mathit{dom}(E) \cap \mathit{RefLoc}$,
- for all $r \in \mathit{dom}(\sigma)$, there exist t and p such that $E(r) = \mathtt{Ref}_p\ t$ and $E\,; p \vdash \sigma(r) : t$,
- for each e' in T, $E\,; \mathtt{P} \vdash e' : \mathtt{Unit}$,
- for each (e, f, a, P) in O, $E\,; \mathtt{P} \vdash e : \mathtt{Unit}$ and $E\,; \mathtt{P} \vdash f : \mathtt{Unit}$, and for each e' in P, $E\,; \mathtt{P} \vdash e' : \mathtt{Unit}$,
- for each $r \in \mathit{dom}(l)$, there exists t such that $E(r) = \mathtt{Ref}_{\mathtt{P}}\ t$ and $E\,; \mathtt{P} \vdash l(r) : t$.

In the special case where O and l are empty, we may omit them and simply say that $\langle \sigma, T \rangle$ is well-typed. This condition is equivalent to $\langle \sigma, T, \mathtt{unit} \rangle$ being well-typed according to the definition of Section 6.2. Thus, whether $\langle \sigma, T \rangle$ is well-typed can be understood and proved entirely in terms of the higher-level definitions, without any regard for optimistic concurrency.

We obtain that typability is preserved by computation ($\longmapsto_{ow}^*$):

THEOREM 8.1 (Preservation of Typability). *If $\langle \sigma, T, O, l \rangle$ is well-typed and $\langle \sigma, T, O, l \rangle \longmapsto_{ow}^* \langle \sigma', T', O', l' \rangle$ then $\langle \sigma', T', O', l' \rangle$ is well-typed.*

$\langle \sigma, T, O.(\mathcal{P}[\,(\lambda x.\ e)\ V\,], f, a, P).O', l\rangle$	$\longmapsto_{ow}$	$\langle \sigma, T, O.(\mathcal{P}[\,e[V/x]\,], f, a, P).O', l\rangle$	(Trans Appl P)$_{ow}$
$\langle \sigma, T.\mathcal{U}[\,(\lambda x.\ e)\ V\,].T', O, l\rangle$	$\longmapsto_{ow}$	$\langle \sigma, T.\mathcal{U}[\,e[V/x]\,].T', O, l\rangle$	(Trans Appl U)$_{ow}$
$\langle \sigma, T, O.(\mathcal{P}[\,\texttt{ref}\ V\,], f, a, P).O', l\rangle$	$\longmapsto_{ow}$	$\langle \sigma[r \mapsto V], T, O.(\mathcal{P}[\,r\,], f, a, P).O', l\rangle$ if $r \in \mathit{RefLoc} - \mathit{dom}(\sigma)$	(Trans Ref P)$_{ow}$
$\langle \sigma, T.\mathcal{U}[\,\texttt{ref}\ V\,].T', O, l\rangle$	$\longmapsto_{ow}$	$\langle \sigma[r \mapsto V], T.\mathcal{U}[\,r\,].T', O, l\rangle$ if $r \in \mathit{RefLoc} - \mathit{dom}(\sigma)$	(Trans Ref U)$_{ow}$
$\langle \sigma, T, O.(\mathcal{P}[\,!r\,], f, a, P).O', l\rangle$	$\longmapsto_{ow}$	$\langle \sigma, T, O.(\mathcal{P}[\,V\,], f, r.a, P).O', l\rangle$ if $\sigma(r) = V$	(Trans Deref P)$_{ow}$
$\langle \sigma, T.\mathcal{U}[\,!r\,].T', O, l\rangle$	$\longmapsto_{ow}$	$\langle \sigma, T.\mathcal{U}[\,V\,].T', O, l\rangle$ if $\sigma(r) = V$	(Trans Deref U)$_{ow}$
$\langle \sigma, T, O.(\mathcal{P}[\,r := V\,], f, a, P).O', l\rangle$	$\longmapsto_{ow}$	$\langle \sigma[r \mapsto V], T, O.(\mathcal{P}[\,\texttt{unit}\,], f, r.a, P).O', l'\rangle$ where $l' =$ if $r \in \mathit{dom}(l)$ then l else $l.[r \mapsto \sigma(r)]$	(Trans Set P)$_{ow}$
$\langle \sigma, T.\mathcal{U}[\,r := V\,].T', O, l\rangle$	$\longmapsto_{ow}$	$\langle \sigma[r \mapsto V], T.\mathcal{U}[\,\texttt{unit}\,].T', O, l\rangle$	(Trans Set U)$_{ow}$
$\langle \sigma, T, O.(\mathcal{P}[\,\texttt{async}\ e\,], f, a, P).O', l\rangle$	$\longmapsto_{ow}$	$\langle \sigma, T, O.(\mathcal{P}[\,\texttt{unit}\,], f, a, e.P).O', l\rangle$	(Trans Async P)$_{ow}$
$\langle \sigma, T.\mathcal{U}[\,\texttt{async}\ e\,].T', O, l\rangle$	$\longmapsto_{ow}$	$\langle \sigma, e.T.\mathcal{U}[\,\texttt{unit}\,].T', O, l\rangle$	(Trans Async U)$_{ow}$
$\langle \sigma, T, O.(\mathcal{P}[\,\texttt{blockUntil}\ \texttt{true}\,], f, a, P).O', l\rangle$	$\longmapsto_{ow}$	$\langle \sigma, T, O.(\mathcal{P}[\,\texttt{unit}\,], f, a, P).O', l\rangle$	(Trans Block P)$_{ow}$
$\langle \sigma, T.\mathcal{U}[\,\texttt{blockUntil}\ \texttt{true}\,].T', O, l\rangle$	$\longmapsto_{ow}$	$\langle \sigma, T.\mathcal{U}[\,\texttt{unit}\,].T', O, l\rangle$	(Trans Block U)$_{ow}$
$\langle \sigma, T, O, l\rangle$	$\longmapsto_{ow}$	$\langle \sigma l, \mathit{origin}(O).T, \emptyset, \emptyset\rangle$	(Trans Undo)$_{ow}$
$\langle \sigma, T, O.(\mathcal{P}[\,\texttt{unprotected}\ e\,], f, a, P).O', l\rangle$	$\longmapsto_{ow}$	$\langle \sigma, T.\mathcal{P}[\,\texttt{unprotected}\ e\,].P, O.O', l - a\rangle$ if $(\mathcal{P}[\,\texttt{unprotected}\ e\,], f, a, P)$ does not conflict with $O.O'$	(Trans Unprotect)$_{ow}$
$\langle \sigma, T, O.(\texttt{unit}, f, a, P).O', l\rangle$	$\longmapsto_{ow}$	$\langle \sigma, T.P, O.O', l - a\rangle$ if $(\texttt{unit}, f, a, P)$ does not conflict with $O.O'$	(Trans Done)$_{ow}$
$\langle \sigma, T.\mathcal{E}[\,\texttt{unprotected}\ V\,].T', O, l\rangle$	$\longmapsto_{ow}$	$\langle \sigma, T.\mathcal{E}[\,V\,].T', O, l\rangle$	(Trans Close)$_{ow}$
$\langle \sigma, T.e.T', O, l\rangle$	$\longmapsto_{ow}$	$\langle \sigma, T.T', (e, e, \emptyset, \emptyset).O, l\rangle$	(Trans Activate)$_{ow}$

Figure 10. Transition rules of the abstract machine, with optimistic concurrency (weak).

In fact, we prove that if $\langle \sigma, T, O, l\rangle$ is well-typed with respect to an environment E, then $\langle \sigma', T', O', l'\rangle$ is well-typed with respect to an extension of E. In the cases of (Trans Ref ...)$_{ow}$, (Trans Deref ...)$_{ow}$, and (Trans Set ...)$_{ow}$, which deal with a reference location of type $\texttt{Ref}_{p_0}\ t_0$, if the transition is protected, then p_0 must be P, and if the transition is unprotected, then p_0 must be U. It follows that, if $\langle \sigma, T, O, l\rangle \longmapsto^*_{ow} \langle \sigma', T', O', l'\rangle$ and $\langle \sigma, T, O, l\rangle$ is well-typed, then there exist subsets P and U of $\mathit{dom}(\sigma')$ such that the protected transitions in $\langle \sigma, T, O, l\rangle \longmapsto^*_{ow} \langle \sigma', T', O', l'\rangle$ allocate, read, or write only reference locations in P, and the unprotected transitions in $\langle \sigma, T, O, l\rangle \longmapsto^*_{ow} \langle \sigma', T', O', l'\rangle$ allocate, read, or write only reference locations in U. Moreover, reference locations reset by (Trans Undo)$_{ow}$ are in P. The subsets in question consist of the reference locations declared with effects P and U, respectively, in the environment.

Using Theorem 8.1, we establish the correctness for the weak semantics with optimistic concurrency with respect to the high-level, strong semantics.

THEOREM 8.2 (Correctness). *Assume that $\langle \sigma, T\rangle$ is well-typed. Consider a computation*

$$\langle \sigma, T, \emptyset, \emptyset\rangle \longmapsto^*_{ow} \langle \sigma', T', \emptyset, \emptyset\rangle$$

Then there is a strong computation

$$\langle \sigma, T, \texttt{unit}\rangle \longmapsto^*_{s} \langle \sigma'', T'', \texttt{unit}\rangle$$

for some σ'' and T'' such that σ' is an extension of σ'' and $T'' = T$ up to reordering.

More generally, in the proof of this result we establish that if $\langle \sigma, T, \emptyset, \emptyset\rangle \longmapsto^*_{ow} \langle \sigma', T', O', l'\rangle$ then there is a strong computation $\langle \sigma, T, \texttt{unit}\rangle \longmapsto^*_{s} \langle \sigma'', T'', \texttt{unit}\rangle$ where $\sigma' l'$ is an extension of σ'', and $T'' = T'.\mathit{origin}(O')$ up to reordering. We also add some further conditions in order to permit an inductive proof.

9. Related Work

This paper is related to work in several areas. There have been informal definitions about how STM or atomic blocks should be used by programmers (Section 9.1); Section 6 is our formalization of these criteria. There have also been several formal semantics for atomic blocks (Section 9.2); our strong semantics is similar to existing definitions, but our weaker semantics go further towards the details of actual implementations. We believe that they are the first to expose problems like those of Section 5. Finally, work on defining weak memory models inspired the approach of considering violation-free programs (Section 9.3).

9.1 Informal Definitions

Several papers propose adding atomic blocks to Java, C#, and similar languages, relying on STM implementations that offer weak atomicity. The criteria for using atomic blocks correctly are usually treated informally (Harris and Fraser 2003; Adl-Tabatabai et al. 2006; Allen et al. 2007). For example, Harris and Fraser (2003) provide a form of separation rule, saying that each shared location should either be protected by a given mutex, or be accessed in atomic blocks, or be marked `volatile`. Our zombie example of Section 5.3 shows the problem with this style of definition: the locations accessed by a zombie transaction depend on the STM implementation, not just on the source language.

Our violation-freedom criterion tries to formalize this definition. This approach may also be applicable to the Fortress language, where Allen et al. (2007) require that "updates to shared locations should always be performed using an atomic expression", or to the extensions to Java in Adl-Tabatabai et al. (2006) that require that "all potentially concurrent accesses to shared memory are properly guarded by atomic regions". Of course, both languages include features not present in the AME calculus, so there may be further subtleties.

The Atomos language notably provides atomic blocks with strong atomicity (Carlstrom et al. 2006). It employs a hardware implementation of transactions.

9.2 Formal Definitions

Jagannathan et al. (2005) define TFJ, an extension to Featherweight Java (Igarashi et al. 2001). They model a source language where transactions include internal fork-join parallelism, and they explore two implementations based on optimistic concurrency control and on two-phase locking. Although steps of the executions of transactions can be interleaved, all TFJ memory accesses are made transactionally so the problems we are studying do not occur.

Liblit (2006) defines a detailed operational semantics for the LogTM hardware. This semantics models the creation and termination of threads, the execution of transactional and non-transactional memory accesses, the interleaving of memory accesses within transactions, and the use of open-nested and closed-nested transactions. The semantics implements strong atomicity. A memory access is not permitted to execute if it would conflict with a concurrent transaction; non-transacted operations are "stalled" until they may run without conflict. Commit and roll-back are both modeled as single transitions. Like Jagannathan et al.'s semantics, Liblit's semantics does not expose the problems that we are studying.

Harris et al. (2005) provide an operational semantics for atomic blocks in Haskell. The semantics is split into three layers: a *core* layer that contains transitions for the evaluation of pure functional code, a *transactional* layer that contains STM operations and pure functional code, and an *I/O* layer that contains input/output operations, pure functional code, and atomic blocks of transactional code. In this semantics, complete transactions execute as single steps in the I/O layer, without interleaving between transactions or between transacted and non-transacted code.

Scott (2006) tackles another aspect of the subject: what is the *sequential specification* of transactional memory as a shared object in Herlihy and Wing's formalism: e.g., what values may a transactional read return, and under what circumstances must a particular transaction commit successfully? Extending Scott's model to consider non-transacted accesses to the same memory would provide another way of approaching the problems of Section 5.

In current work, Moore and Grossman (2008) are studying the operational semantics of transactions. Our studies were started independently, but we have had the opportunity to compare notes. While it appears that our high-level goals and our techniques are similar, there are a number of differences in our results. In particular, Moore and Grossman focus on traditional atomic blocks, with internal concurrency but with no yielding and no provision for unprotected fragments except at the top level; they have yet to analyze schemes with optimistic concurrency. Despite these and other differences, the works are consistent in their demonstrating the viability and value of precise operational semantics for the constructs considered.

9.3 Memory Models

Adve and Hill (1990) introduced the idea of providing strong semantics to programs that obey a set of formally specified constraints; our definition and use of violation-freedom is partly inspired by their approach. Spear et al. (2007) independently identified the link between this work and transactional memory, proposing a hierarchy of models for sharing data between transactional and non-transactional code.

In many languages the memory model must also consider programs that are *not* correctly synchronized so that, for example, a programmer cannot use data races to violate safety and security properties of a virtual machine. Grossman et al. (2006) have started to examine some of the questions that arise when extending this aspect of a memory model to programs written with atomic blocks.

Blundell et al. (2005) illustrate how a program may run to completion under a particular implementation of transactions, but will always deadlock under strong atomicity. Their example is not violation-free and not well-typed under a type system like that of Section 6.2. Furthermore, with some implementations of weak atomicity (Harris and Fraser 2003), the example will never run to completion. This point is yet another illustration of how semantics with weak atomicity are tied to the details of particular implementations.

10. Conclusion and Further Work

The present exploration of language constructs represents the foundation for ongoing work on programming with transactional memory. Understanding the semantics of the constructs and the related tradeoffs has proven both challenging and worthwhile. In particular, the realization that weak semantics like that of Section 8 *do not* correctly execute all violation-free programs indicates that implementation techniques employed in Bartok-STM *cannot* be used without further language restrictions or other precautions.

We have demonstrated that imposing a strong language restriction, static separation of mutable state, lets us give the programmer the attractive behavior of the strong semantics even with a very permissive implementation. In hindsight, this fact may not appear surprising, but it is worth noting that several definitions of separation are possible (e.g., (Harris and Fraser 2003; Harris et al. 2005; Moore and Grossman 2008)), and that they have substantially different consequences; for instance, some definitions do not suffice in the presence of zombies (see Section 9). Although separation is appealing in a functional setting, it is probably less palatable in an imperative language where most data is considered mutable, and would therefore require marshaling across the separation boundary. These results suggest a number of directions for future work—by developing the type system (to allow more programs to be correctly typed), the language constructs (perhaps to describe data transfer between protected and unprotected modes), or the STM implementation (perhaps to support more programs with the strong semantics). This exploration highlights the benefits of co-design of these three aspects of the language and its implementation.

We have also explored several alternative semantics. Clearly there are many others. Some of those that capture appealing implementation strategies may be worth studying further. Moreover, incorporating some of the subtleties of relaxed memory models may lead to further problems and assumptions.

In addition to the type system in this paper, we have developed and analyzed a type system that characterizes "yielding" behavior. With this type system, the caller of a function obtains static information on whether the function may yield and therefore commit. Combining the two type systems is straightforward, and may be attractive if yielding and separation are generalized (so, for example, yielding may commit only a part of the heap).

Our initial exploration of AME includes writing example programs. At this point, we have confidence that the constructs are interesting and useful, and in any case we expect that some of the ideas and results of our work will be of value whether or not particular constructs are widely adopted. Designing constructs and designing languages are distinct activities; further research should inform a language design based on AME.

Acknowledgements

We would like to thank Dan Grossman for interesting discussions on our work, Virendra Marathe for identifying the original form of the "privatization" problem during an internship at Microsoft, Dave Detlefs for encouraging us to study it, and Simon Peyton-Jones and Jean-Philippe Martin for comments on earlier drafts of this paper.

References

Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, 2006.

Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–37, 2006.

Sarita V. Adve and Mark D. Hill. Weak ordering – a new definition. *SIGARCH Comput. Archit. News*, 18(3a):2–14, 1990.

Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *Proc. 2002 USENIX Annual Technical Conference (USENIX-02)*, pages 289–302, 2002.

Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress language specification, v1.0β. Technical report, 2007.

Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *Proc. 2005 Workshop on Duplicating, Deconstructing and Debunking*, 2005.

Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The Atomos transactional programming language. In *PLDI '06: Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–13, 2006.

David Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

Dan Grossman, Jeremy Manson, and William Pugh. What do high-level memory models mean for transactions? In *MSPC '06: Proc. 2006 Workshop on Memory System Performance and Correctness*, pages 62–69, 2006.

Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proc. 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, 2003.

Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proc. 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60, 2005.

Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *PLDI '06: Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 14–25, 2006.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

Michael Isard and Andrew Birrell. Automatic mutual exclusion. In *Proc. 11th Workshop on Hot Topics in Operating Systems*, 2007.

Suresh Jagannathan, Jan Vitek, Adam Welc, and Antony Hosking. A transactional object calculus. *Science of Computer Programming*, 57 (2):164–186, 2005.

Bradley C. Kuszmaul and Charles E. Leiserson. Transactions everywhere. Technical report, 2003. http://hdl.handle.net/1721.1/3692.

Ben Liblit. An operational semantics for LogTM. Technical Report 1571, U. Wisconsin–Madison, 2006. Version 1.0.

Jeremy Manson, Jason Baker, Antonio Cunei, Suresh Jagannathan, Marek Prochazka, Bin Xin, and Jan Vitek. Preemptible atomic regions for real-time Java. In *Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS)*, 2005a.

Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *POPL '05: Proc. 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 378–391, 2005b.

Katherine F. Moore and Dan Grossman. High-level small-step operational semantics for transactions. To appear, *POPL '08: Proc. 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008.

Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *Proc. 12th International Symposium on High-Performance Computer Architecture*, pages 254–265. 2006.

Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *PLDI '06: Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, 2006.

Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proc. Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–197, 2006.

Michael L. Scott. Sequential specification of transactional memory semantics. In *Proc. 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. 2006.

Nir Shavit and Dan Touitou. Software transactional memory. In *Proc. 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in STM. In *PLDI '07: Proc. 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 78–88, 2007.

Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. Technical Report #915, Computer Science Department, University of Rochester, 2007.

Nicholas Sterling. Warlock: A static data race analysis tool. In *Proc. USENIX Winter Technical Conference*, 1993.

Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe futures for Java. In *OOPSLA '05: Proc. 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 439–453, 2005.