Semantics of Transactional Memory and Automatic Mutual Exclusion

MARTÍN ABADI
Microsoft Research, Silicon Valley
and
University of California, Santa Cruz

ANDREW BIRRELL
Microsoft Research, Silicon Valley

TIM HARRIS
Microsoft Research, Cambridge

and

MICHAEL ISARD
Microsoft Research, Silicon Valley

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

Additional Key Words and Phrases: Atomicity, correctness

Author's address: ....
A preliminary version of this paper was presented at the 35th Annual ACM SIGPLAN – SIGACT Symposium on Principles of Programming Languages (POPL 2008).

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, languages provide 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].

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. 2006], 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. Concretely, we present our results focusing on the AME programming model (Section 2). We use AME for two reasons. First, while developing this new programming model, we hope to avoid the pitfalls encountered in earlier work; 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). We review a series of examples from earlier work, and show how they can behave when implemented on some STMs [Blundell et al. 2006; Shpeisman et al. 2007]. We argue that many of these examples are incorrectly synchronized, but that others, informally, do not contain race conditions.

We then examine two language restrictions that enable weaker semantics to implement the strong semantics (Section 6). First, we consider a *violation-freedom* condition, which formalizes the sense in which some of the examples of Section 5 are race-free. For programs that satisfy this condition, we show that a semantics with weak atomicity and in-place update implements the strong semantics (Section 7). We obtain an analogous result for a lower-level semantics with roll-back (Section 8). 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. 2005]. Our second language restriction is a type system that statically separates data according to whether or not it is accessed transactionally. With practical STM implementations in mind, we study a semantics with weak atomicity, in-place update, roll-back, and also concurrent execution of transactions and lazy conflict detection. We show that, for well-typed programs, this semantics implements the strong semantics (Section 9). Violation-freedom does not suffice for this property.

We discuss related work in Section 10. We conclude in Section 11 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. An appendix contains proofs.

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. Other recent papers describe aspects of our further research on AME, our implementation, and our experience so far [Abadi 2008; Abadi et al. 2008; Abadi et al. 2008; 2009].

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 programs 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 in 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 serializa-

tion 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].

In AME, we avoid the "stack ripping" by allowing 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 provide the following construct:

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

Fig. 1. Syntax of the AME calculus.

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

The current atomic fragment ends before the unprotected statement, and a new one starts after it.

The AME programming model does not stipulate how unprotected code is written. In practice, we expect this code to be expressed in a language such as C#, and to use the mechanisms for synchronization that exist in such a language.

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 (ef), allocation (`ref` e, which allocates a new reference location and returns it after initializing it to 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 particular, unprotected code can be written in this syntax, rather than in some legacy language (cf. Section 2.4). In unprotected contexts, `async` e will behave roughly like "fork e", and `blockUntil` e will behave roughly like "wait until e". The precise meaning of this and other constructions is defined uniformly 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'.

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

Fig. 2. State space.

—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:

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

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, 8, and 9 we consider richer and weaker semantics that add these features.

We give two formulations of the strong semantics. They share a definition of states (Section 4.1), but differ in their definitions of the transition relation (Sections 4.2 and 4.3). We compare them in Section 4.4. In short, they are basically equivalent, but the first is simpler and should be easier to understand, while the second is closer to lower-level semantics and therefore more convenient for the technical development that follows.

Throughout this paper, we prefer not to focus on the subtleties of unprotected code, and in particular on the semantics of unprotected code with race conditions. An abundant literature addresses this subject. We therefore rely on simple models of memory, with strong guarantees for unprotected code, focusing on the interaction between protected and unprotected code.

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,

$$\mathcal{E} \;=\; [\,] \mid \mathcal{E}\ e \mid V\ \mathcal{E} \mid \texttt{ref}\ \mathcal{E} \mid\ !\mathcal{E} \mid \mathcal{E} := e \mid r := \mathcal{E} \mid \texttt{blockUntil}\ \mathcal{E} \mid \texttt{unprotected}\ \mathcal{E}$$

Fig. 3. Evaluation contexts.

—a distinguished active expression e.

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.

We identify expressions with threads of computation. The semantics does not describe stacks or other thread-specific data. Informally, the pool is intended to hold pending work while computation progresses in the active expression. The formulations of the strong semantics differ on what computations may take place in the active expression, and on whether any computations may be done directly in the pool.

4.2 Steps (First Formulation)

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.

The next possible operation in an expression is found by decomposing the expression into an evaluation context and a subexpression that describes this operation. We use the evaluation contexts defined in Figure 3. 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 $\mathcal{C}$.

Figure 4 specifies a transition relation $\longmapsto$ 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". Rules (Trans Appl), (Trans Ref), (Trans Deref), and (Trans Set) are the standard rules for call-by-value function application and for manipulating reference stores. 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. Rule (Trans Async) shows how an asynchronous computation is forked: an expression is added to the pool. Rule (Trans Block) reduces `blockUntil true`. Rules (Trans Unprotect) and (Trans Close) move the active expression to the pool when it is of the form $\mathcal{E}[\ \texttt{unprotected}\ e\ ]$; in the special case where e is a value, (Trans Close) offers the possibility of removing the occurrence of `unprotected`. Finally, rule (Trans Activate) takes a new active

(Trans Appl)
$\langle \sigma, T, \mathcal{E}[\ (\lambda x.\ e)\ V\ ] \rangle \longmapsto \langle \sigma, T, \mathcal{E}[\ e[V/x]\ ] \rangle$

(Trans Ref)
$\langle \sigma, T, \mathcal{E}[\ \texttt{ref}\ V\ ] \rangle \longmapsto \langle \sigma[r \mapsto V], T, \mathcal{E}[\ r\ ] \rangle$
if $r \in \mathit{RefLoc} - \mathit{dom}(\sigma)$

(Trans Deref)
$\langle \sigma, T, \mathcal{E}[\ !r\ ] \rangle \longmapsto \langle \sigma, T, \mathcal{E}[\ V\ ] \rangle$
if $\sigma(r) = V$

(Trans Set)
$\langle \sigma, T, \mathcal{E}[\ r := V\ ] \rangle \longmapsto \langle \sigma[r \mapsto V], T, \mathcal{E}[\ \texttt{unit}\ ] \rangle$

(Trans Async)
$\langle \sigma, T, \mathcal{E}[\ \texttt{async}\ e\ ] \rangle \longmapsto \langle \sigma, e.T, \mathcal{E}[\ \texttt{unit}\ ] \rangle$

(Trans Block)
$\langle \sigma, T, \mathcal{E}[\ \texttt{blockUntil true}\ ] \rangle \longmapsto \langle \sigma, T, \mathcal{E}[\ \texttt{unit}\ ] \rangle$

(Trans Unprotect)
$\langle \sigma, T, \mathcal{E}[\ \texttt{unprotected}\ e\ ] \rangle \longmapsto \langle \sigma, T.\mathcal{E}[\ \texttt{unprotected}\ e\ ], \texttt{unit} \rangle$

(Trans Close)
$\langle \sigma, T, \mathcal{E}[\ \texttt{unprotected}\ V\ ] \rangle \longmapsto \langle \sigma, T.\mathcal{E}[\ V\ ], \texttt{unit} \rangle$

(Trans Activate)
$\langle \sigma, T.e.T', \texttt{unit} \rangle \longmapsto \langle \sigma, T.T', e \rangle$

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

expression from the pool.

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

$$\langle \sigma, T, \mathcal{E}[\ \texttt{yield}\ ] \rangle \longmapsto \langle \sigma, T.\mathcal{E}[\ \texttt{unit}\ ], \texttt{unit} \rangle$$

This rule is easily derived from (Trans Close) of Figure 4 and the definition of `yield` as `unprotected unit`.

As in most other semantics, sometimes no further evaluation is appropriate in an expression, and the rules—or their absence—reflect this fact. In particular, there are no rules to evaluate `unit` and `blockUntil false` because, intuitively, evaluation should be complete in the former case and stuck in the latter. Because expressions can have multiple decompositions, however, one should be careful in understanding the possible applications of rules. In particular, although no rule explicitly addresses expressions of the form $\mathcal{E}[\ \texttt{unit}\ ]$, those expressions are not all alike:

—In some cases, computation is simply complete, as when $\mathcal{E} = [\ ]$.

—In some cases, computation is stuck, as when $\mathcal{E} = [\ ]e$, since `unit` is not a function.

—Other times, $\mathcal{E}[\ \texttt{unit}\ ]$ can be decomposed in multiple ways. For instance, $(\lambda x.\ t)$ `unit` can be decomposed as either $\mathcal{E}_1[\ \texttt{unit}\ ]$ where $\mathcal{E}_1 = (\lambda x.\ t)\ [\ ]$ or as $\mathcal{E}_2[\ (\lambda x.\ t)\ \texttt{unit}\ ]$ where $\mathcal{E}_2 = [\ ]$; the latter decomposition allows further

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

Fig. 5. Additional evaluation contexts for strong semantics.

progress in the computation.

4.3 Steps (Second Formulation)

According to the first formulation of the strong semantics, all substantial work—both transacted and not—happens in the active expression. Next we give an alternative definition of steps that embodies an execution model in which the pool is not completely passive.

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. Strong atomicity means that expressions in the pool can take steps only when the active expression is `unit`; thus, it restricts the interleaving of steps of "protected" and "unprotected" work.

In order to formalize these ideas, we use additional kinds of evaluation contexts, defined in Figure 5:

—$\mathcal{P}$ evaluation contexts are for the execution of protected code: the position for evaluation is not under `unprotected`.

—$\mathcal{U}$ evaluation contexts are for the execution of unprotected code: the position for evaluation is under `unprotected`.

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

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

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

Figure 6 gives rules that specify a new transition relation. These rules resemble those of Section 4.2, except that "unprotected" work happens in the pool rather than in the active expression. The subscript s in $\longmapsto_s$ indicates that this is a strong semantics. (We use other subscripts in similar symbols below. We do not put a subscript on the symbol $\longmapsto$ in Section 4.2 because $\longmapsto$ is not used beyond Section 4.)

4.4 Comparing the Two Formulations

The definitions of Figures 4 and 6 have evident superficial differences. The former treats protected and unprotected computation steps more uniformly; therefore, it is more elegant and we prefer it as a simple statement of the semantics of the AME

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

Fig. 6. Transition rules of the abstract machine (strong, alternative version).

calculus. The latter draws a sharper distinction between protected and unprotected computation steps; in this respect, it matches better what happens in lower-level semantics, below.

Despite these superficial differences, the definitions are almost equivalent. Formally, we have:

PROPOSITION 4.1. —*If* $\langle \sigma, T, e \rangle \longmapsto \langle \sigma', T', e' \rangle$ *then* $\langle \sigma, T, e \rangle \longmapsto_s^* \langle \sigma', T', e' \rangle$.
—*If* $\langle \sigma, T, e \rangle \longmapsto_s \langle \sigma', T', e' \rangle$ *then there exists* T'' *such that* $\langle \sigma, T, e \rangle \longmapsto^* \langle \sigma', T'', e' \rangle$ *and* $T'' = T'$ *up to reordering.*

(We write $\longmapsto^*$ and $\longmapsto_s^*$ for the reflexive-transitive closures of $\longmapsto$ and $\longmapsto_s$, respectively.)

Below we rely exclusively on the transition relation $\longmapsto_s$ of Figure 6. Via Proposition 4.1, results about $\longmapsto_s$ immediately translate into results about $\longmapsto$.

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 in AME 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 [Shpeisman et al. 2007; Schneider et al. 2008; Bronson et al. 2009].

In this section we discuss the ways in which 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.

Blundell et al. [2006], Grossman et al. [2006], Shpeisman et al. [2007], and Menon et al. [2008] describe further examples, and examine their behavior when these examples are implemented with different TM techniques.

In this paper, however, we focus on the problems that occur on the Bartok-STM system [Harris et al. 2006], which is the basis of our present implementation of AME. Bartok-STM is based on a whole-program ahead-of-time optimizing compiler for C#. Within atomic sections, the compiler introduces calls to the Bartok-STM runtime system to perform concurrency control and logging on each access to the heap. The Bartok-STM runtime system is implemented in C# (albeit using low-level, unsafe features). This allows parts of the runtime system, including the STM implementation, to be inlined into application code. Harris et al. [2006] provide further details.

In the Bartok-STM runtime system, transactional 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-STM [Saha et al. 2006], because of their efficiency on many practical workloads.

5.1 Unsynchronized Access to Shared Locations

The first set of examples, in Figure 7, comes from work by Shpeisman et al. [2007] on implementing strong atomicity. In the figure, code is protected (i.e., runs transactionally) unless it is contained in an `unprotected` block.

None of these examples is implemented with strong semantics on Bartok-STM. The NR and ILU problems occur with Bartok-STM because conflicts between protected reads and unprotected writes are not detected. The SLU problem occurs with Bartok-STM because the implementation of roll-backs can overwrite concurrent non-transactional stores. The IDR and SDR problems occur with Bartok-STM because in-place updates allow the unprotected code to see intermediate writes by a transaction—whether or not the transaction eventually commits.

One may reasonably ask "Do these problems matter?" because they intuitively involve data races. However, other examples are less clear-cut, and some 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.

We present three of these race-free examples in Sections 5.2–5.4. For concreteness, again, we focus on Bartok-STM; however, variants of these examples affect many STM systems that allow data to be shared between protected and unprotected code.

(NR) Non-repeatable reads: Can we obtain `r1!=r2`*?*

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

(ILU) Intermediate lost updates: Can we obtain `x==1`*?*

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

(SLU) Speculative lost updates: Can we obtain `x==0`*?*

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

(IDR) Intermediate dirty reads: Can we obtain `r1==1`*?*

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

(SDR) Speculative dirty reads: Can we obtain `x==0, y==1`*?*

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

Fig. 7. Example data races based on those of Shpeisman et al.. All fields initially hold 0. Registers `r1` and `r2` are thread-local.

5.2 Zombie Transactions

This 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 [Wang et al. 2007]. 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`.

5.3 Privatization

Another example illustrates a *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 5.2 in that it occurs because `A2` continues to execute as a zombie.

5.4 Publication

A final example illustrates a *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 reordered 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. 2005]. 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 many 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. These criteria provide precise definitions, in the AME calculus, of whether or not a program is correctly synchronized.

—The first criterion, violation-freedom, says that, dynamically, data cannot be accessed with and without protection at the same time. Violation-freedom is inspired by conditions for data-race-free programming, such as those of Adve and Hill [1990]. Violation-freedom lets us say, formally, that the examples of Section 5.1 have data races, and that the examples of Sections 5.2–5.4 are correctly synchronized.

—The second criterion, separation, is embodied in a static discipline that guarantees that protected and unprotected computations do not use the same reference locations. Separation is inspired by the type system of Haskell-STM which distinguishes ordinary mutable state from transactionally mutable state [Harris et al. 2005]. As we prove, separation implies violation-freedom. None of the examples in Sections 5.1–5.4 is correctly synchronized under this stronger criterion.

In Sections 7, 8, and 9, 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.

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.

For example, this discipline does not allow the following two programs:

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

and

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

In the former program, the unprotected code $y := !x$ and the protected code $x :=$ `true` lead to obviously conflicting accesses to x. The latter program is somewhat subtler. There, the protected code $x :=$ `true` will be executed within a transaction that starts with the vacuous statement `blockUntil true`. Under the strong semantics, no unprotected computation can happen while the transaction is under way, even if the unprotected computation is otherwise ready to proceed. Nevertheless, according to our definitions, this program does generate a violation, because the strong semantics leads to a state of the form $\langle \sigma, \mathcal{U}[\ r' := !r\ ], \mathcal{P}[\ r :=$ `true` $\ ] \rangle$, where r and r' are the locations that x and y represent, respectively.

Violations are distinct from traditional race conditions in that violations do not consider conflicts between unprotected computations. However, race conditions can be defined in similar terms (e.g., [Abadi et al. 2006]). For this purpose, one may say that a state $\langle \sigma, e_1. \cdots .e_n, e \rangle$ has a race if $e_i = \mathcal{U}[\ f\ ]$ for some $i = 1..n$, $e_j = \mathcal{U}'[\ f'\ ]$ for some $j = 1..n$, $i \neq j$, f and f' are reads or writes on the same location, and at least one of them is a write. Such conflicts, while often harmful, are not violations according to our definitions.

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 8, using judgments and rules for reasoning about the judgments. The core of the type system is the set of rules for the judgment $E\,; p \vdash e : t$, which is 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. While this sort of judgment is common in the literature on "type-and-effect systems", our particular choice of effects is specific to our system. When p is `P`, this judgment means that the evaluation of e accesses only the part of the reference store for protected computations; when p is `U`, this judgment 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, $s \rightarrow^p t$ 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.

The following small example illustrates the restrictions that the type system

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 8. The first-order type system for separation.

imposes:

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

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:

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

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

—$dom(\sigma) = dom(E) \cap \mathit{RefLoc}$,

—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 $E \vdash \langle \sigma, e_1.\cdots.e_n, e\rangle$ for some E. 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 $\mathcal{P}[$* `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. In particular, separation seems less fragile in the presence of compiler optimizations: whereas it may be hard to predict whether a program transformation could introduce violations, it seems easier to tell whether it could break separation.

7. WEAK SEMANTICS

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. This semantics may however be regarded as the strongest of weak semantics. It embodies Single-Global-Lock Atomicity [Menon et al. 2008]. In a straightforward, inefficient implementation of this semantics, a single global lock would be used for mutual exclusion between atomic fragments. Unprotected code would not access this global lock, and may execute concurrently with atomic fragments. This semantics still serializes transactions: only one piece of protected code can run at a time. We present it because it is an instructive, simple illustration of weak semantics, not because of direct resemblance to actual implementations. We consider concurrency between transactions in Section 9.

7.1 States

The states of this weak semantics are basically the same as those of the strong semantics, as defined in Section 4.1. However, only for Section 7, we make the following small technical restriction on expressions: in any expression of the form `async` e, any occurrences of `unprotected` are under a λ. This technical restriction basically ensures that an unprotected computation is not the first thing that happens in an asynchronous computation. Thus, using our abbreviations, we can write `async` $(unit;$ `unprotected` $e')$, but not `async` $($`unprotected` $e')$. More generally, we can write `async` $(unit; e')$, for any e'. So the restriction, while convenient for

$$\mathcal{G} \;=\; T.\mathcal{U}.T', e' \mid T, \mathcal{P}$$

Fig. 9. Additional evaluation contexts for weak semantics.

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

Fig. 10. Transition rules of the abstract machine (weak).

some of our formal definitions and proofs, has only a minor impact on the power of the language.

7.2 Steps

Figures 9 and 10 define this weak semantics. Figure 9 introduces evaluation contexts $\mathcal{G}$, defined so that $\mathcal{G}[\ e\ ]$ includes both $T.\mathcal{U}[\ e\ ].T', e'$ and $T, \mathcal{P}[\ e\ ]$. Since $\mathcal{G}[\ e\ ]$ is either $T.\mathcal{U}[\ e\ ].T', e'$ or $T, \mathcal{P}[\ e\ ]$, we write $e_0.\mathcal{G}[\ e_1\ ]$ as an abbreviation for $e_0.T.\mathcal{U}[\ e_1\ ].T', e'$ or $e_0.T, \mathcal{P}[\ e_1\ ]$, respectively. Figure 10 defines the transition relation $\longmapsto_w$, very much like Figure 6 defines the transition relation $\longmapsto_s$ above. The rules of Figure 10 are directly analogous to those with corresponding names in Figure 6. Unlike the relation $\longmapsto_s$, however, the relation $\longmapsto_w$ allows unprotected computations to proceed even when the active expression is not `unit`. Formally, this difference arises because $\mathcal{G}[\ e\ ]$ may be of the form $T.\mathcal{U}[\ e\ ].T', e'$ when e' is not `unit`, while $\mathcal{F}[\ e\ ]$ would be of this form only if e' is `unit`.

Consider a transition $\langle \sigma, e_1.\cdots.e_n, e\rangle \longmapsto_w \langle \sigma', e'_1.\cdots.e'_{n'}, e'\rangle$. Unless it is an instance of (Trans Unprotect)$_w$ or (Trans Activate)$_w$, the transition is defined in terms of a context that has a hole either in $e_1.\cdots.e_n$ and in $e'_1.\cdots.e'_{n'}$ or in e and in e'. We say that the transition is protected if the hole is in e and in e', and say that the transition is unprotected if the hole is in $e_1.\cdots.e_n$ and in $e'_1.\cdots.e'_{n'}$. By definition, transitions that that are instances of (Trans Close)$_w$ are always unprotected; transitions that are instances of (Trans Unprotect)$_w$ or (Trans Activate)$_w$ are neither protected nor unprotected. We have $\langle \sigma, T, e\rangle \longmapsto_s \langle \sigma', T', e'\rangle$ when $\langle \sigma, T, e\rangle \longmapsto_w \langle \sigma', T', e'\rangle$ and, if this transition is unprotected, then $e = \texttt{unit}$.

7.3 Correctness

The goal of this section is to establish the correctness of the weak semantics as an implementation of the simpler strong semantics, assuming the absence of violations.

As the examples of Section 5 suggest, the violation-freedom hypothesis is needed. For example, the program

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

which does not obey the hypothesis, may set y to `true` under the weak semantics but not under the strong semantics.

We obtain the following theorem:

THEOREM 7.1. *Assume that all strong computations that start from the state* $\langle \sigma, T, \texttt{unit}\rangle$ *are violation-free. Consider a weak computation* $\langle \sigma, T, \texttt{unit}\rangle \longmapsto_w^* \langle \sigma', T', e'\rangle$. *Then there is a strong computation* $\langle \sigma, T, \texttt{unit}\rangle \longmapsto_s^* \langle \sigma', T', e'\rangle$.

By Corollary 6.3, we immediately derive that if $\langle \sigma, T, \texttt{unit}\rangle$ is well-typed and $\langle \sigma, T, \texttt{unit}\rangle \longmapsto_w^* \langle \sigma', T', e'\rangle$, then $\langle \sigma, T, \texttt{unit}\rangle \longmapsto_s^* \langle \sigma', T', e'\rangle$.

8. WEAK SEMANTICS WITH ROLL-BACK

Refining the semantics of Section 7, in this section we define a semantics that models eager versioning, in which transactions make in-place updates to the heap and are rolled back if they abort for some reason. We show that this weak semantics is correct for violation-free programs.

Although this weak semantics still serializes transactions, it is 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. 2005]. We focus on the roll-backs that one may wish to perform when a computation is blocked on a false condition (that is, on roll-backs caused when a thread decides to abort and retry later). Other kinds of roll-backs (in particular, those used in the implementation of optimistic concurrency schemes) may be treated similarly, with the possible addition of machinery for conflict detection. We consider those in Section 9. While the two kinds of roll-backs have somewhat different motivations, and the latter can be more intricate, they have substantial semantic commonalities: the corresponding rules are quite similar.

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

Fig. 11. State space, with roll-back.

8.1 States

Figure 11 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 $r \in \mathit{RefLoc}$ occurs free in $\sigma(r')$, in T, in e, in f, in l, or in P, then $r \in \mathit{dom}(\sigma)$. 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 $\mathit{dom}(l)$ be the set of locations r for which l is defined, and when $r \in \mathit{dom}(l)$ we write $l(r)$ for the value V to which r is mapped.

8.2 Steps

Figure 12 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.
Formally, the rule (Trans Block `false` Undo)$_{rw}$ shows how one takes the last element of the log and applies it to the reference store, and the rule (Trans Block `false` Restore)$_{rw}$, shows how the origin expression is added back to the pool when the log is empty.

—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].
Formally, this detail is apparent in that no rule logs allocations, so neither (Trans Block `false` Undo)$_{rw}$ nor any other rule could revert the allocations.

—No undo facilities are provided for unprotected computations.

$$
\begin{array}{ll}
(\text{Trans Appl P})_{rw} & \\
\quad \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 \\
 & \\
(\text{Trans Appl U})_{rw} & \\
\quad \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 \\
 & \\
(\text{Trans Ref P})_{rw} & \\
\quad \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 \\
 & \quad \text{if } r \in \mathit{RefLoc} - \mathit{dom}(\sigma) \\
(\text{Trans Ref U})_{rw} & \\
\quad \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 \\
 & \quad \text{if } r \in \mathit{RefLoc} - \mathit{dom}(\sigma) \\
(\text{Trans Deref P})_{rw} & \\
\quad \langle \sigma, T, \mathcal{P}[\ !r\ ], f, l, P\rangle & \longmapsto_{rw}\ \langle \sigma, T, \mathcal{P}[\ V\ ], f, l, P\rangle \\
 & \quad \text{if } \sigma(r) = V \\
(\text{Trans Deref U})_{rw} & \\
\quad \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 \\
 & \quad \text{if } \sigma(r) = V \\
(\text{Trans Set P})_{rw} & \\
\quad \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 \\
 & \quad \text{where } l' = \text{if } r \in \mathit{dom}(l) \text{ then } l \text{ else } l.[r \mapsto \sigma(r)] \\
(\text{Trans Set U})_{rw} & \\
\quad \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 \\
 & \\
(\text{Trans Async P})_{rw} & \\
\quad \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 \\
 & \\
(\text{Trans Async U})_{rw} & \\
\quad \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 \\
 & \\
(\text{Trans Block } \texttt{true} \text{ P})_{rw} & \\
\quad \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 \\
 & \\
(\text{Trans Block } \texttt{true} \text{ U})_{rw} & \\
\quad \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 \\
 & \\
(\text{Trans Block } \texttt{false} \text{ Restore})_{rw} & \\
\quad \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 \\
 & \\
(\text{Trans Block } \texttt{false} \text{ Undo})_{rw} & \\
\quad \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 \\
 & \\
(\text{Trans Unprotect})_{rw} & \\
\quad \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 \\
 & \\
(\text{Trans Done})_{rw} & \\
\quad \langle \sigma, T, \texttt{unit}, f, l, P\rangle & \longmapsto_{rw}\ \langle \sigma, T.P, \texttt{unit}, \texttt{unit}, \emptyset, \emptyset\rangle \\
 & \\
(\text{Trans Close})_{rw} & \\
\quad \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 \\
 & \\
(\text{Trans Activate})_{rw} & \\
\quad \langle \sigma, T.e.T', \texttt{unit}, \texttt{unit}, \emptyset, \emptyset\rangle & \longmapsto_{rw}\ \langle \sigma, T.T', e, e, \emptyset, \emptyset\rangle
\end{array}
$$

Fig. 12. Transition rules of the abstract machine, with roll-back (weak).

—Since this is a weak semantics, unprotected computations may be interleaved with protected computations, and even with the roll-backs of protected computations.
—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.

8.3 Correctness

Much as in Section 7.3, 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.

For this purpose, we consider an intermediate strong semantics with roll-back. Figure 13 defines the transition relation of the intermediate semantics. Its rules are straightforward variants of those of Figure 12. Basically, the rules for unprotected computation in Figure 13 demand that no protected computation be in progress, while those of Figure 12 do not impose this requirement. For instance, (Trans Ref U)$_{rs}$ applies to states of the form $\langle \sigma, T.\mathcal{U}[\ \texttt{ref}\ V\ ].T', \texttt{unit}, \texttt{unit}, \emptyset, \emptyset\rangle$, while (Trans Ref U)$_{rw}$ applies to states of the more general form $\langle \sigma, T.\mathcal{U}[\ \texttt{ref}\ V\ ].T', e, f, l, P\rangle$.

The intermediate semantics implements the strong semantics without roll-back:

LEMMA 8.1. *If* $\langle \sigma, T, \texttt{unit}, \texttt{unit}, \emptyset, \emptyset\rangle \longmapsto_{rs}^{*} \langle \sigma', T', \texttt{unit}, \texttt{unit}, \emptyset, \emptyset\rangle$*, then there exists* σ'' *and* T'' *such that* σ' *extends* σ''*,* $T'' = T'$ *up to reordering, and* $\langle \sigma, T, \texttt{unit}\rangle \longmapsto_{s}^{*} \langle \sigma'', T'', \texttt{unit}\rangle$.

Note that this result does not require violation-freedom.

Furthermore, the weak semantics with roll-back implements the intermediate semantics, in the following sense:

LEMMA 8.2. *Assume that all strong computations with roll-back that start from the state* $\langle \sigma, T, \texttt{unit}, \texttt{unit}, \emptyset, \emptyset\rangle$ *are violation-free. Consider a weak computation with roll-back* $\langle \sigma, T, \texttt{unit}, \texttt{unit}, \emptyset, \emptyset\rangle \longmapsto_{rw}^{*} \langle \sigma', T', e', f', l', P'\rangle$*. Then there is a strong computation with roll-back* $\langle \sigma, T, \texttt{unit}, \texttt{unit}, \emptyset, \emptyset\rangle \longmapsto_{rs}^{*} \langle \sigma', T', e', f', l', P'\rangle$.

Here, we do rely on a violation-freedom hypothesis. The definition of violation-freedom for the intermediate semantics is almost exactly that of Section 6.1, but in terms of $\longmapsto_{rs}$ rather than $\longmapsto_{s}$, and ignoring the additional state components of the intermediate semantics (origin, log, and pending threads).

A final lemma relates the definitions of violation-freedom:

LEMMA 8.3. *Assume that all strong computations that start from the state* $\langle \sigma, T, \texttt{unit}\rangle$ *are violation-free. Then all strong computations with roll-back that start from the state* $\langle \sigma, T, \texttt{unit}, \texttt{unit}, \emptyset, \emptyset\rangle$ *are violation-free.*

Combining Lemmas 8.1, and 8.2, and 8.3, we obtain the following theorem:

THEOREM 8.4 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

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

(Trans Appl P)$_{rs}$
$\langle \sigma, T, \mathcal{P}[\ (\lambda x.\ e)\ V\ ], f, l, P\rangle \longmapsto_{rs} \langle \sigma, T, \mathcal{P}[\ e[V/x]\ ], f, l, P\rangle$

(Trans Appl U)$_{rs}$
$\langle \sigma, T.\mathcal{U}[\ (\lambda x.\ e)\ V\ ].T', \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset\rangle \longmapsto_{rs} \langle \sigma, T.\mathcal{U}[\ e[V/x]\ ].T', \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset\rangle$

(Trans Ref P)$_{rs}$
$\langle \sigma, T, \mathcal{P}[\ \mathtt{ref}\ V\ ], f, l, P\rangle \longmapsto_{rs} \langle \sigma[r \mapsto V], T, \mathcal{P}[\ r\ ], f, l, P\rangle$
if $r \in \mathit{RefLoc} - \mathit{dom}(\sigma)$

(Trans Ref U)$_{rs}$
$\langle \sigma, T.\mathcal{U}[\ \mathtt{ref}\ V\ ].T', \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset\rangle \longmapsto_{rs} \langle \sigma[r \mapsto V], T.\mathcal{U}[\ r\ ].T', \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset\rangle$
if $r \in \mathit{RefLoc} - \mathit{dom}(\sigma)$

(Trans Deref P)$_{rs}$
$\langle \sigma, T, \mathcal{P}[\ !r\ ], f, l, P\rangle \longmapsto_{rs} \langle \sigma, T, \mathcal{P}[\ V\ ], f, l, P\rangle$
if $\sigma(r) = V$

(Trans Deref U)$_{rs}$
$\langle \sigma, T.\mathcal{U}[\ !r\ ].T', \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset\rangle \longmapsto_{rs} \langle \sigma, T.\mathcal{U}[\ V\ ].T', \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset\rangle$
if $\sigma(r) = V$

(Trans Set P)$_{rs}$
$\langle \sigma, T, \mathcal{P}[\ r := V\ ], f, l, P\rangle \longmapsto_{rs} \langle \sigma[r \mapsto V], T, \mathcal{P}[\ \mathtt{unit}\ ], f, l', P\rangle$
where $l' = \text{if } r \in \mathit{dom}(l) \text{ then } l \text{ else } l.[r \mapsto \sigma(r)]$

(Trans Set U)$_{rs}$
$\langle \sigma, T.\mathcal{U}[\ r := V\ ].T', \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset\rangle \longmapsto_{rs} \langle \sigma[r \mapsto V], T.\mathcal{U}[\ \mathtt{unit}\ ].T', \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset\rangle$

(Trans Async P)$_{rs}$
$\langle \sigma, T, \mathcal{P}[\ \mathtt{async}\ e\ ], f, l, P\rangle \longmapsto_{rs} \langle \sigma, T, \mathcal{P}[\ \mathtt{unit}\ ], f, l, e.P\rangle$

(Trans Async U)$_{rs}$
$\langle \sigma, T.\mathcal{U}[\ \mathtt{async}\ e\ ].T', \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset\rangle \longmapsto_{rs} \langle \sigma, e.T.\mathcal{U}[\ \mathtt{unit}\ ].T', \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset\rangle$

(Trans Block `true` P)$_{rs}$
$\langle \sigma, T, \mathcal{P}[\ \mathtt{blockUntil}\ \mathtt{true}\ ], f, l, P\rangle \longmapsto_{rs} \langle \sigma, T, \mathcal{P}[\ \mathtt{unit}\ ], f, l, P\rangle$

(Trans Block `true` U)$_{rs}$
$\langle \sigma, T.\mathcal{U}[\ \mathtt{blockUntil}\ \mathtt{true}\ ].T', \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset\rangle \longmapsto_{rs} \langle \sigma, T.\mathcal{U}[\ \mathtt{unit}\ ].T', \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset\rangle$

(Trans Block `false` Restore)$_{rs}$
$\langle \sigma, T, \mathcal{P}[\ \mathtt{blockUntil}\ \mathtt{false}\ ], f, \emptyset, P\rangle \longmapsto_{rs} \langle \sigma, f.T, \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset\rangle$

(Trans Block `false` Undo)$_{rs}$
$\langle \sigma, T, \mathcal{P}[\ \mathtt{blockUntil}\ \mathtt{false}\ ], f, l.[r \mapsto V], P\rangle \longmapsto_{rs} \langle \sigma[r \mapsto V], T, \mathcal{P}[\ \mathtt{blockUntil}\ \mathtt{false}\ ], f, l, P\rangle$

(Trans Unprotect)$_{rs}$
$\langle \sigma, T, \mathcal{P}[\ \mathtt{unprotected}\ e\ ], f, l, P\rangle \longmapsto_{rs} \langle \sigma, T.P.\mathcal{P}[\ \mathtt{unprotected}\ e\ ], \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset\rangle$

(Trans Done)$_{rs}$
$\langle \sigma, T, \mathtt{unit}, f, l, P\rangle \longmapsto_{rs} \langle \sigma, T.P, \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset\rangle$

(Trans Close)$_{rs}$
$\langle \sigma, T.\mathcal{E}[\ \mathtt{unprotected}\ V\ ].T', \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset\rangle \longmapsto_{rs} \langle \sigma, T.\mathcal{E}[\ V\ ].T', \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset\rangle$

(Trans Activate)$_{rs}$
$\langle \sigma, T.e.T', \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset\rangle \longmapsto_{rs} \langle \sigma, T.T', e, e, \emptyset, \emptyset\rangle$

Fig. 13. Transition rules of the abstract machine, with roll-back (strong).

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 de-allocated. 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.

As in Section 7.3, we immediately derive a correctness property based on typing by Corollary 6.3: if $\langle \sigma, T, \texttt{unit} \rangle$ is well-typed and $\langle \sigma, T, \texttt{unit}, \texttt{unit}, \emptyset, \emptyset \rangle \longmapsto_{rw}^{*} \langle \sigma', T', \texttt{unit}, \texttt{unit}, \emptyset, \emptyset \rangle$, then $\langle \sigma, T, \texttt{unit} \rangle \longmapsto_{s}^{*} \langle \sigma'', T'', \texttt{unit} \rangle$ for some σ'' and T'' such that σ' is an extension of σ'' and $T'' = T'$ up to reordering.

9. 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 using weak atomicity, in-place updates to the reference store, and lazy conflict detection. Alternative definitions (for instance, a strong semantics with optimistic concurrency, buffered updates, and eager conflict detection) seem viable too, but generally less interesting from our perspective and easier to analyze.

Optimistic concurrency raises further correctness issues. Interestingly, and unlike for our semantics of Sections 7 and 8, violation-freedom is *not* a sufficient condition for correctness in this case. As in our examples of Section 5.2–5.3, a program can be violation-free under the strong semantics but have lower-level violations because of zombie transactions. (The example of Section 5.2 is violation-free because the statement `x = 42` is not reachable under the strong semantics. In the example of Section 5.3 the statements `x ++` and `x=42` cannot conflict under the strong semantics. Similarly, in the example of Section 5.4, the statements `x = 42` and `r1 = x` cannot conflict under the strong semantics.) 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.

9.1 States

As described in Figure 14, 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.

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

Fig. 14. State space, with optimistic concurrency.

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 exposition. Even in the present form, the semantics exhibits challenging features.

9.2 Steps

Figure 15 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 $\mathit{origin}(O)$ 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. Thus, only the rules that represent commits ((Trans Done)$_{ow}$ and (Trans Unprotect)$_{ow}$) demand the absence of conflicts.

For simplicity, the rules do not include some secondary features sufficiently explored in the semantics with roll-back of Section 8. 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`). Thus, the rule (Trans Undo)$_{ow}$ applies an entire log l to a reference store σ, and applies without any condition on the protected computations (O) in progress.

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

(Trans Appl P)$_{ow}$
$\langle \sigma, T, O.(\mathcal{P}[\ (\lambda x.\ e)\ V\ ], f, a, P).O', l\rangle \longmapsto_{ow} \langle \sigma, T, O.(\mathcal{P}[\ e[V/x]\ ], f, a, P).O', l\rangle$

(Trans Appl U)$_{ow}$
$\langle \sigma, T.\mathcal{U}[\ (\lambda x.\ e)\ V\ ].T', O, l\rangle \longmapsto_{ow} \langle \sigma, T.\mathcal{U}[\ e[V/x]\ ].T', O, l\rangle$

(Trans Ref P)$_{ow}$
$\langle \sigma, T, O.(\mathcal{P}[\ \texttt{ref}\ V\ ], f, a, P).O', l\rangle \longmapsto_{ow} \langle \sigma[r \mapsto V], T, O.(\mathcal{P}[\ r\ ], f, a, P).O', l\rangle$
if $r \in \mathit{RefLoc} - \mathit{dom}(\sigma)$

(Trans Ref U)$_{ow}$
$\langle \sigma, T.\mathcal{U}[\ \texttt{ref}\ V\ ].T', O, l\rangle \longmapsto_{ow} \langle \sigma[r \mapsto V], T.\mathcal{U}[\ r\ ].T', O, l\rangle$
if $r \in \mathit{RefLoc} - \mathit{dom}(\sigma)$

(Trans Deref P)$_{ow}$
$\langle \sigma, T, O.(\mathcal{P}[\ !r\ ], f, a, P).O', l\rangle \longmapsto_{ow} \langle \sigma, T, O.(\mathcal{P}[\ V\ ], f, r.a, P).O', l\rangle$
if $\sigma(r) = V$

(Trans Deref U)$_{ow}$
$\langle \sigma, T.\mathcal{U}[\ !r\ ].T', O, l\rangle \longmapsto_{ow} \langle \sigma, T.\mathcal{U}[\ V\ ].T', O, l\rangle$
if $\sigma(r) = V$

(Trans Set P)$_{ow}$
$\langle \sigma, T, O.(\mathcal{P}[\ r := V\ ], f, a, P).O', l\rangle \longmapsto_{ow} \langle \sigma[r \mapsto V], T, O.(\mathcal{P}[\ \texttt{unit}\ ], f, r.a, P).O', l'\rangle$
where $l' = \text{if } r \in \mathit{dom}(l) \text{ then } l \text{ else } l.[r \mapsto \sigma(r)]$

(Trans Set U)$_{ow}$
$\langle \sigma, T.\mathcal{U}[\ r := V\ ].T', O, l\rangle \longmapsto_{ow} \langle \sigma[r \mapsto V], T.\mathcal{U}[\ \texttt{unit}\ ].T', O, l\rangle$

(Trans Async P)$_{ow}$
$\langle \sigma, T, O.(\mathcal{P}[\ \texttt{async}\ e\ ], f, a, P).O', l\rangle \longmapsto_{ow} \langle \sigma, T, O.(\mathcal{P}[\ \texttt{unit}\ ], f, a, e.P).O', l\rangle$

(Trans Async U)$_{ow}$
$\langle \sigma, T.\mathcal{U}[\ \texttt{async}\ e\ ].T', O, l\rangle \longmapsto_{ow} \langle \sigma, e.T.\mathcal{U}[\ \texttt{unit}\ ].T', O, l\rangle$

(Trans Block P)$_{ow}$
$\langle \sigma, T, O.(\mathcal{P}[\ \texttt{blockUntil true}\ ], f, a, P).O', l\rangle \longmapsto_{ow} \langle \sigma, T, O.(\mathcal{P}[\ \texttt{unit}\ ], f, a, P).O', l\rangle$

(Trans Block U)$_{ow}$
$\langle \sigma, T.\mathcal{U}[\ \texttt{blockUntil true}\ ].T', O, l\rangle \longmapsto_{ow} \langle \sigma, T.\mathcal{U}[\ \texttt{unit}\ ].T', O, l\rangle$

(Trans Undo)$_{ow}$
$\langle \sigma, T, O, l\rangle \longmapsto_{ow} \langle \sigma l, \mathit{origin}(O).T, \emptyset, \emptyset\rangle$

(Trans Unprotect)$_{ow}$
$\langle \sigma, T, O.(\mathcal{P}[\ \texttt{unprotected}\ e\ ], f, a, P).O', l\rangle \longmapsto_{ow} \langle \sigma, T.\mathcal{P}[\ \texttt{unprotected}\ e\ ].P, O.O', l - a\rangle$
if $(\mathcal{P}[\ \texttt{unprotected}\ e\ ], f, a, P)$ does not conflict with $O.O'$

(Trans Done)$_{ow}$
$\langle \sigma, T, O.(\texttt{unit}, f, a, P).O', l\rangle \longmapsto_{ow} \langle \sigma, T.P, O.O', l - a\rangle$
if $(\texttt{unit}, f, a, P)$ does not conflict with $O.O'$

(Trans Close)$_{ow}$
$\langle \sigma, T.\mathcal{E}[\ \texttt{unprotected}\ V\ ].T', O, l\rangle \longmapsto_{ow} \langle \sigma, T.\mathcal{E}[\ V\ ].T', O, l\rangle$

(Trans Activate)$_{ow}$
$\langle \sigma, T.e.T', O, l\rangle \longmapsto_{ow} \langle \sigma, T.T', (e, e, \emptyset, \emptyset).O, l\rangle$

Fig. 15. Transition rules of the abstract machine, with optimistic concurrency (weak).

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 $(\text{Trans} \ldots \text{P})_{ow}$ are always protected;

—transitions that are instances of $(\text{Trans} \ldots \text{U})_{ow}$ or of $(\text{Trans Close})_{ow}$ are always unprotected;

—transitions that are instances of $(\text{Trans Undo})_{ow}$, $(\text{Trans Unprotect})_{ow}$, $(\text{Trans Done})_{ow}$, or $(\text{Trans Activate})_{ow}$ are neither protected nor unprotected.

9.3 Correctness

As explained above, the absence of high-level violations does not in general suffice for correctness. The following small program (from [Abadi et al. 2008]) illustrates this point in the context of the AME calculus, thus confirming that problems presented in Section 5 have formal counterparts:

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

Under the strong semantics, execution of the code $x := \texttt{false}; \texttt{blockUntil } !x; y := \texttt{true}$ never reaches $y := \texttt{true}$, so y can never be set to `true`, so z can never be set to `true` either. On the other hand, with optimistic concurrency, z may be set to `true` if the execution of $x := \texttt{true}$ is interleaved between those of $x := \texttt{false}$ and $\texttt{blockUntil } !x$, then y is set `true`, and z is also set to `true` before the conflict on x is detected.

It is plausible that the absence of lower-level violations would suffice for correctness. Such a 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

—$dom(\sigma) = dom(E) \cap \mathit{RefLoc}$,

—for all $r \in dom(\sigma)$, there exist t and p such that $E(r) = \texttt{Ref}_p\ t$ and $E\,; p \vdash \sigma(r) : t$,

—for each e' in T, $E\,; \texttt{P} \vdash e' : \texttt{Unit}$,

—for each (e, f, a, P) in O, $E\,; \texttt{P} \vdash e : \texttt{Unit}$ and $E\,; \texttt{P} \vdash f : \texttt{Unit}$, and for each e' in P, $E\,; \texttt{P} \vdash e' : \texttt{Unit}$,

—for each $r \in dom(l)$, there exists t such that $E(r) = \texttt{Ref}_\texttt{P}\ t$ and $E\,; \texttt{P} \vdash l(r) : t$.

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 $\langle \sigma, T, \texttt{unit} \rangle$ 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 9.1 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.*

In fact, we obtain 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. Moreover, there exist subsets P and U of $dom(\sigma')$ 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. Similarly, 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 9.1, we establish the correctness for the weak semantics with optimistic concurrency with respect to the high-level, strong semantics.

THEOREM 9.2 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 $T'' = T'.origin(O')$ up to reordering. We also add some further conditions in order to permit an inductive proof.

10. RELATED WORK

This paper is related to work in several areas. Work on defining weak memory models inspired the approach of considering violation-free programs (Section 10.1). There have been informal definitions about how STM or atomic blocks should be used by programmers (Section 10.2); Section 6 is our formalization of these criteria. There have also been several formal semantics for atomic blocks (Section 10.3); our strong semantics is similar to existing definitions, but our weaker semantics go further towards the details of actual implementations. We believe that our semantics are the first to formally expose problems like those of Section 5.

10.1 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.

Concurrently with our work, Spear et al. [2007] identified the link between this approach and transactional memory, proposing a hierarchy of models for sharing data between transactional and non-transactional code, and barrier operations to separate transactional and non-transactional uses of objects. In subsequent work, Spear et al. [2008] describe annotations to control whether or not sections of non-transactional code are ordered with respect to transactions. Dalessandro and Scott [2009] describe a "transactional data-race free" (TDRF) programming model. Informally, TDRF appears similar to violation-freedom in combination with ordinary data-race freedom. Dalessandro and Scott argue that a TM with strong atomicity provides little benefit over weaker implementations for non-TDRF programs.

Nevertheless, semantics for languages such as Java and C# must consider what guarantees to provide for programs with data races, as well as to race-free programs. 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. The work of Menon et al. [2008] considers a series of semantics that provide various guarantees for atomic blocks. Menon et al. relate these to the Java memory model, and to a range of lock-based programming idioms that typically do not work with STM-based implementations of atomic blocks. Menon et al. methodically show how to constrain an STM to implement forms of these idioms correctly, even in cases where the lock-based idiom involves an apparent data race.

10.2 Informal Definitions

Several papers propose informal criteria for using atomic blocks in Java, C#, and similar languages [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`. Adl-Tabatabai et al. [2006] require that "all potentially concurrent accesses to shared memory are properly guarded by atomic regions". In the Fortress language, Allen et al. [2007] require that "updates to shared locations should always be performed using an atomic expression".

These informal definitions can be problematic. While it is clear that some examples do not satisfy them (e.g., NR from Figure 7), it is ambiguous whether examples like the privatization idiom from Section 5.3 are meant to be allowed. The zombie example of Section 5.2 illustrates a further problem with such definitions: the locations accessed by a zombie transaction depend on the STM implementation, not just on the source program.

Several researchers identified related difficulties and proposed mechanisms for addressing them. Thus, Dice and Shavit [2006] describe the need to let transactional stores to an object "drain" from the system when memory passes in and out of transactional usage. Dice et al. [2006] introduce the term "zombie transaction" to refer to a transaction that has observed an inconsistent read-set but has yet to abort. Wang et al. [2007] describe a privatization example and the idea of waiting for concurrent transactions to become quiescent before accessing privatized data non-transactionally. The particular, succinct privatization idiom that we present in Section 5.3 was brought to our attention by Marathe.

10.3 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.

Moore and Grossman [2008] also study the operational semantics of transactions. While our high-level goals and our techniques are similar, our studies were started independently, and 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.

Other recent work is addressing the problem of verifying implementations of transactional memory [Cohen et al. 2007; Guerraoui et al. 2008a; 2008b; Tasiran 2008]. This work generally does not consider the programming languages built on top of transactional memory. However, it should be possible to leverage this work in the study of programming-language implementations.

11. 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 9 *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 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 10). 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 further work—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, which we are currently pursuing [Abadi et al. 2008; Abadi et al. 2008; 2009], 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 [Isard and Birrell 2007; Abadi 2008]. 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.

APPENDIX

The appendix contains proofs for the results stated in the body of the paper and some auxiliary results. A few routine arguments are omitted for brevity.

A.1 Proofs for Section 4

Proof of Proposition 4.1. The proofs for both directions are fairly straightforward arguments by cases.

—Let us assume that $\langle \sigma, T, e \rangle \longmapsto \langle \sigma', T', e' \rangle$. We wish to prove that $\langle \sigma, T, e \rangle \longmapsto_s^*$

$\langle \sigma', T', e' \rangle$.

By definition of $\longmapsto$, e is `unit` or of the form $\mathcal{E}[\ f\ ]$, and in the latter case e is of the form $\mathcal{P}[\ f\ ]$ or $\mathcal{U}[\ f\ ]$.

—If e is of the form $\mathcal{P}[\ f\ ]$, then $T = T'$ and immediately $\langle \sigma, T, e \rangle \longmapsto_s \langle \sigma', T', e' \rangle$.

—If e is of the form $\mathcal{U}[\ f\ ]$, we argue by cases on which rule is applied in $\langle \sigma, T, e \rangle \longmapsto \langle \sigma', T', e' \rangle$.

—(Trans Appl), (Trans Ref), (Trans Deref), (Trans Set), (Trans Async), (Trans Block):

$$\begin{array}{rl} \langle \sigma, T, e \rangle & \longmapsto_s \ \langle \sigma', T'.e, \texttt{unit} \rangle \\ & \longmapsto_s \ \langle \sigma', T.e', \texttt{unit} \rangle \\ & \longmapsto_s \ \langle \sigma', T, e' \rangle \end{array}$$

by (Trans Unprotect)$_s$, the corresponding rule (one of (Trans Appl)$_s$, (Trans Ref)$_s$, (Trans Deref)$_s$, (Trans Set)$_s$, (Trans Async)$_s$, (Trans Block)$_s$), and (Trans Activate)$_s$.

—(Trans Unprotect): $T' = T.e$, $e' = \texttt{unit}$, and $\langle \sigma, T, e \rangle \longmapsto_s \langle \sigma', T.e, \texttt{unit} \rangle$ by (Trans Unprotect)$_s$, so $\langle \sigma, T, e \rangle \longmapsto_s \langle \sigma', T', e' \rangle$.

—(Trans Close): e is of the form $\mathcal{E}[\ \texttt{unprotected}\ V\ ]$, $T' = T.\mathcal{E}[\ V\ ]$, $e' = \texttt{unit}$, and

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

by (Trans Unprotect)$_s$ and (Trans Close)$_s$, so $\langle \sigma, T, e \rangle \longmapsto_s^* \langle \sigma', T', e' \rangle$.

—(Trans Activate): Since e is of the form $\mathcal{U}[\ f\ ]$, this rule cannot apply.

—If e is `unit`, then $\langle \sigma, T, e \rangle \longmapsto \langle \sigma', T', e' \rangle$ by (Trans Activate), so $\langle \sigma, T, e \rangle \longmapsto_s \langle \sigma', T', e' \rangle$ by (Trans Activate)$_s$.

—Let us assume that $\langle \sigma, T, e \rangle \longmapsto_s \langle \sigma', T', e' \rangle$. We wish to prove that $\langle \sigma, T, e \rangle \longmapsto^* \langle \sigma', T'', e' \rangle$ for some $T'' = T'$ up to reordering. We proceed by cases on which rule is applied in $\langle \sigma, T, e \rangle \longmapsto_s \langle \sigma', T', e' \rangle$.

—(Trans Appl)$_s$, (Trans Ref)$_s$, (Trans Deref)$_s$, (Trans Set)$_s$, (Trans Async)$_s$, (Trans Block)$_s$: Consider a transition $\langle \sigma, T, e \rangle \longmapsto_s \langle \sigma', T', e' \rangle$ by one of these rules. The transition is defined in terms of a context that has a hole either in T and in T' or in e and in e'.

—In the former case, $e = \texttt{unit}$, T is of the form $T_1.f.T_2$, T' is of the form $T_1.f'.T_2$, the hole is under `unprotected` in f and f', and

$$\begin{array}{rl} \langle \sigma, T, \texttt{unit} \rangle & \longmapsto \ \langle \sigma, T_1.T_2, f \rangle \\ & \longmapsto \ \langle \sigma, T_1.T_2, f' \rangle \\ & \longmapsto \ \langle \sigma', T_1.T_2.f', \texttt{unit} \rangle \end{array}$$

by (Trans Activate), the corresponding rule (one of (Trans Appl), (Trans Ref), (Trans Deref), (Trans Set), (Trans Async), (Trans Block)), and (Trans Unprotect), so $\langle \sigma, T, e \rangle \longmapsto^* \langle \sigma', T_1.T_2.f', e' \rangle$ and $T_1.T_2.f' = T'$ up to reordering.

—In the latter case, $\langle \sigma, T, e \rangle \longmapsto \langle \sigma', T', e' \rangle$ directly by the corresponding rule.

—(Trans Unprotect)$_s$: e is of the form $\mathcal{P}[\ \texttt{unprotected}\ f\ ]$, $e' = \texttt{unit}$, and $T' = T.e$, so $\langle \sigma, T, e \rangle \longmapsto \langle \sigma', T', e' \rangle$ by (Trans Unprotect).

—(Trans Close)$_s$: $e = e' = \mathtt{unit}$, T is of the form $T_1.\mathcal{E}[\ \mathtt{unprotected}\ V\ ].T_2$, T' is $T_1.\mathcal{E}[\ V\ ].T_2$, and

$$\begin{array}{rl}\langle \sigma, T_1.\mathcal{E}[\ \mathtt{unprotected}\ V\ ].T_2, \mathtt{unit}\rangle & \longmapsto \langle \sigma, T_1.T_2, \mathcal{E}[\ \mathtt{unprotected}\ V\ ]\rangle \\ & \longmapsto \langle \sigma, T_1.T_2.\mathcal{E}[\ V\ ], \mathtt{unit}\rangle\end{array}$$

by (Trans Activate) and (Trans Close), so $\langle \sigma, T, e\rangle \longmapsto^* \langle \sigma', T_1.T_2.\mathcal{E}[\ V\ ], e'\rangle$ and $T_1.T_2.\mathcal{E}[\ V\ ] = T'$ up to reordering.

—(Trans Activate)$_s$: $e = \mathtt{unit}$ and T is of the form $T_1.e'.T_2$, so $\langle \sigma, T, e\rangle \longmapsto \langle \sigma', T', e'\rangle$ by (Trans Activate).

□

A.2 Proofs for Section 6

A first auxiliary result is a replacement lemma, in the style of Wright and Felleisen [1994]. It immediately extends to typing states $\langle \sigma, e_1.\cdots.e_n, e\rangle$.

LEMMA A.1 REPLACEMENT. *Consider a derivation $\mathcal{D}$ of $E\,;p \vdash \mathcal{E}[\ e_0\ ] : t$. Assume that this derivation includes, as a subderivation, a proof $\mathcal{D}_0$ of the judgment $E\,;p_0 \vdash e_0 : t_0$ for the occurrence of e_0 in $\mathcal{E}[\ \cdot\ ]$. Assume that we also have a derivation $\mathcal{D}'_0$ of $E\,;p_0 \vdash e'_0 : t_0$ for some e'_0. Let $\mathcal{D}'$ be obtained from $\mathcal{D}$ by replacing $\mathcal{D}_0$ with $\mathcal{D}'_0$, and e_0 with e'_0 in $\mathcal{E}$. Then $\mathcal{D}'$ is a derivation of $E\,;p \vdash \mathcal{E}[\ e'_0\ ] : t$.*

The next results say that values can be typed with either effect, if they can be typed at all.

LEMMA A.2. *If $E\,;p \vdash V : t$ then $E\,;q \vdash V : t$.*

The next result is a standard substitution lemma, restricted to values.

LEMMA A.3 VALUE SUBSTITUTION. *If $E, x : s, E'\,;p \vdash e : t$ and $E\,;q \vdash V : s$ then $E, E'\,;p \vdash e[V/x] : t$.*

PROOF. As usual, the proof is by induction on the derivation of $E, x : s, E'\,;p \vdash e : t$. In the case of (Exp x), we rely on Lemma A.2. □

Another lemma deals with updates to the state.

LEMMA A.4. *Let $r \in dom(\sigma)$ and $E(r) = \mathtt{Ref}_{p_0}\ t_0$. If $E \vdash \langle \sigma, e_1.\cdots.e_n, e\rangle$ and $E\,;p_0 \vdash V : t_0$, then $E \vdash \langle \sigma[r \mapsto V], e_1.\cdots.e_n, e\rangle$.*

The final lemma is a syntactic analysis of contexts.

LEMMA A.5. *If $E \vdash \langle \sigma, T, \mathcal{P}[\ e\ ]\rangle$ then there exists t such that $E\,;\mathtt{P} \vdash e : t$. If $E \vdash \langle \sigma, T.\mathcal{U}[\ e'\ ].T', e\rangle$ then there exists t such that $E\,;\mathtt{U} \vdash e' : t$.*

PROOF. Since a state can be typed only if all of its components can be typed, if $E \vdash \langle \sigma, T, \mathcal{P}[\ e\ ]\rangle$ then there exist t and p such that $E\,;p \vdash e : t$, and if $E \vdash \langle \sigma, T.\mathcal{U}[\ e'\ ].T', e\rangle$ then there exist t and p such that $E\,;p \vdash e' : t$. It remains to prove that p must be $\mathtt{P}$ in the former case and $\mathtt{U}$ in the latter case.

Generalizing from $\mathtt{Unit}$ to an arbitrary t', we show that if $E\,;\mathtt{P} \vdash \mathcal{P}[\ e\ ] : t'$ then $E\,;\mathtt{P} \vdash e : t$ for some t, and that if $E\,;\mathtt{P} \vdash \mathcal{U}[\ e'\ ] : t'$ then $E\,;\mathtt{U} \vdash e' : t$ for some t. We show in addition that if $E\,;\mathtt{U} \vdash \mathcal{E}[\ e'\ ] : t'$ then $E\,;\mathtt{U} \vdash e' : t$ for some t.

The proofs are by induction on the forms of $\mathcal{P}[\ e\ ]$, $\mathcal{U}[\ e'\ ]$, and $\mathcal{E}[\ e'\ ]$.

—When $\mathcal{P}[\ e\ ]$ is e, $E\,;\mathtt{P} \vdash \mathcal{P}[\ e\ ] : \mathtt{Unit}$ is $E\,;\mathtt{P} \vdash e : \mathtt{Unit}$. When $\mathcal{P}[\ e\ ]$ is of another form, the typing derivation of $E\,;\mathtt{P} \vdash \mathcal{P}[\ e\ ] : \mathtt{Unit}$ must rely on a proof of $E\,;\mathtt{P} \vdash \mathcal{P}'[\ e\ ] : \mathtt{Unit}$ for some smaller $\mathcal{P}'[\ e\ ]$, according to the typing rules.

—Similarly, when $\mathcal{E}[\ e'\ ]$ is e', we are done immediately, and the cases for other forms are immediate applications of the induction hypothesis to smaller contexts.

—When $\mathcal{U}[\ e'\ ]$ is $\mathtt{unprotected}\ \mathcal{E}[\ e'\ ]$, we have $E\,;\mathtt{U} \vdash \mathcal{E}[\ e'\ ] : \mathtt{Unit}$, and the previous case applies. For other forms of $\mathcal{U}[\ e'\ ]$, the cases are immediate applications of the induction hypothesis to smaller contexts.

□

Proof of Theorem 6.1. We prove that if $\langle \sigma, e_1.\cdots.e_n, e\rangle$ is well-typed and $\langle \sigma, e_1.\cdots.e_n, e\rangle \longmapsto_s \langle \sigma', e'_1.\cdots.e'_{n'}, e'\rangle$ then $\langle \sigma', e'_1.\cdots.e'_{n'}, e'\rangle$ is well-typed. The first claim follows immediately by induction.

The proof is by cases on the operational-semantics rule being applied. In each case, we show that if $E \vdash \langle \sigma, e_1.\cdots.e_n, e\rangle$ then $E' \vdash \langle \sigma', e'_1.\cdots.e'_{n'}, e'\rangle$, where, unless indicated otherwise, $E' = E$ and $n' = n$. In several cases, we consider the typings of certain subexpressions that occur in evaluation contexts; those typings are with respect to E, since the holes in the contexts are never under binders.

—(Trans Appl)$_s$: The typing of $\langle \sigma, \mathcal{F}[\ (\lambda x.\, e)\ V\ ]\rangle$ must rely on (Exp Appl) and (Exp Fun). Specifically, we must have $E\,;p_0 \vdash (\lambda x.\, e)\ V : t_0$ for some t_0 and p_0, and therefore $E\,;p_0 \vdash \lambda x.\, e : t_1 \to^{p_0} t_0$ and $E\,;p_0 \vdash V : t_1$ for some t_1, and therefore $E, x : t_1\,;p_0 \vdash e : t_0$. By Lemma A.3, we obtain $E\,;p_0 \vdash e[V/x] : t_0$. By Lemma A.1, we obtain a typing of $\langle \sigma, \mathcal{F}[\ e[V/x]\ ]\rangle$.

—(Trans Ref)$_s$: The typing of $\langle \sigma, \mathcal{F}[\ \mathtt{ref}\ V\ ]\rangle$ must rely on (Exp Ref). Specifically, we must have $E\,;p_0 \vdash \mathtt{ref}\ V : \mathtt{Ref}_{p_0}\ t_0$ for some t_0 and p_0, and therefore $E\,;p_0 \vdash V : t_0$. We extend E with $r : \mathtt{Ref}_{p_0}\ t_0$. We can do this extension because $r \in \mathit{RefLoc} - \mathit{dom}(\sigma)$, hence $r \notin \mathit{dom}(E)$. By a weakening (adding $r : \mathtt{Ref}_{p_0}\ t_0$ to E for typing $\langle \sigma, \mathcal{F}[\ \mathtt{ref}\ V\ ]\rangle$) and Lemma A.1, we obtain a typing of $\langle \sigma, \mathcal{F}[\ r\ ]\rangle$.

—(Trans Deref)$_s$: The typing of $\langle \sigma, \mathcal{F}[\ !r\ ]\rangle$ must rely on (Exp Deref). Specifically, we must have $E\,;p_0 \vdash !r : t_0$ for some t_0 and p_0, and therefore $E\,;p_0 \vdash r : \mathtt{Ref}_{p_0}\ t_0$. Since r is a variable, its type must come from the environment E, so by hypothesis $E\,;p_0 \vdash V : t_0$ where $V = \sigma(r)$. By Lemma A.1, we obtain a typing for $\langle \sigma, \mathcal{F}[\ V\ ]\rangle$.

—(Trans Set)$_s$: The typing of $\langle \sigma, \mathcal{F}[\ r := V\ ]\rangle$ must rely on (Exp Set). Specifically, we must have $E\,;p_0 \vdash r := V : \mathtt{Unit}$ for some p_0, and therefore $E\,;p_0 \vdash V : t_0$ and $E\,;p_0 \vdash r : \mathtt{Ref}_{p_0}\ t_0$ for some p_0. By Lemma A.1, we can transform a typing of $\langle \sigma, \mathcal{F}[\ r := V\ ]\rangle$ into a typing of $\langle \sigma, \mathcal{F}[\ \mathtt{unit}\ ]\rangle$, and since $E\,;p_0 \vdash V : t_0$ and $E(r) = \mathtt{Ref}_{p_0}\ t_0$, we also obtain a typing of $\langle \sigma[r \mapsto V], \mathcal{F}[\ \mathtt{unit}\ ]\rangle$ by Lemma A.4.

—(Trans Async)$_s$: The typing of $\langle \sigma, \mathcal{F}[\ \mathtt{async}\ e\ ]\rangle$ must rely on (Exp Async). Specifically, we must have $E\,;p_0 \vdash \mathtt{async}\ e : \mathtt{Unit}$ for some p_0, and therefore that $E\,;\mathtt{P} \vdash e : \mathtt{Unit}$. By Lemma A.1, we can transform a typing of $\langle \sigma, \mathcal{F}[\ \mathtt{async}\ e\ ]\rangle$ into a typing of $\mathcal{F}[\ \mathtt{unit}\ ]$, and then into a typing of $\langle \sigma, e.\mathcal{F}[\ \mathtt{unit}\ ]\rangle$, letting $n' = n + 1$.

—(Trans Block)$_s$: The typing of $\langle \sigma, \mathcal{F}[\ \mathtt{blockUntil\ true}\ ]\rangle$ must rely on (Exp Block), specifically on a derivation of $E\,;p_0 \vdash \mathtt{blockUntil\ true} : \mathtt{Unit}$ for some

p_0. By Lemma A.1, we obtain a typing of $\langle \sigma, \mathcal{F}[\ \texttt{unit}\ ] \rangle$.

—(Trans Unprotect)$_s$: This case requires a trivial rearrangement in the typing, with $n' = n + 1$.

—(Trans Close)$_s$: The typing of $\langle \sigma, T.\mathcal{E}[\ \texttt{unprotected}\ V\ ].T', e' \rangle$ must rely on (Exp Unprotect). Specifically, we must have $E\,; p_0 \vdash \texttt{unprotected}\ V : t_0$ for some t_0 and p_0, and $E\,; \texttt{U} \vdash V : t_0$, so $E\,; p_0 \vdash V : t_0$ by Lemma A.2. By Lemma A.1, we obtain a typing of $\mathcal{E}[\ V\ ]$ and then of $\langle \sigma, T.\mathcal{E}[\ V\ ].T', e' \rangle$.

—(Trans Activate)$_s$: This case requires a trivial rearrangement in the typing, with $n' = n - 1$.

□

Examining the proof, we note that if $\langle \sigma, T, e \rangle$ is well-typed with respect to an environment E, then $\langle \sigma', T', e' \rangle$ is well-typed with respect to an extension of E. In the cases of (Trans Ref)$_s$, (Trans Deref)$_s$, and (Trans Set)$_s$, which deal with a reference location of type $\texttt{Ref}_{p_0}\ t_0$, we also note that if the transition is protected, then p_0 must be P, and if the transition is unprotected, then p_0 must be U, by Lemma A.5. It follows that, if $\langle \sigma, T, e \rangle \longmapsto_s^* \langle \sigma', T', e' \rangle$ and $\langle \sigma, T, e \rangle$ is well-typed, then there exist subsets P and U of $dom(\sigma')$ such that the protected transitions in $\langle \sigma, T, e \rangle \longmapsto_s^* \langle \sigma', T', e' \rangle$ allocate, read, or write only reference locations in P, and the unprotected transitions in $\langle \sigma, T, e \rangle \longmapsto_s^* \langle \sigma', T', e' \rangle$ allocate, read, or write only reference locations in U. The subsets in question consist of the reference locations declared with effects P and U, respectively, in the environment.

Proof of Theorem 6.2. This proof is very similar to the proof of the progress theorem in [Abadi 2008]. Indeed, it follows from a slight generalization of that proof to the calculus with `unprotected` (which is not treated in [Abadi 2008]). We omit the details. □

Proof of Corollary 6.3. Suppose that $\langle \sigma, T, e \rangle \longmapsto_s^* \langle \sigma', T', e' \rangle$. By Theorem 6.1, if $\langle \sigma, T, e \rangle$ is well-typed then so is $\langle \sigma', T', e' \rangle$, with some typing environment E'. If there is a violation on a location r in $\langle \sigma', T', e' \rangle$, then T' includes an expression of the form $\mathcal{U}[\ f\ ]$ and e' is of the form $\mathcal{P}[\ f'\ ]$, where f and f' are reads or writes on r. By Lemma A.5, there exists t such that $E'\,; \texttt{U} \vdash f : t$ and t' such that $E'\,; \texttt{P} \vdash f' : t'$. So, according to the typing rules, r must have a type of the form $\texttt{Ref}_\texttt{U}\ t_0$ in E' and also a type of the form $\texttt{Ref}_\texttt{P}\ t_0$ in E'. This conclusion contradicts the well-formedness of E'. □

A.3 Proofs for Section 7

We prove a commutation lemma:

Lemma A.6. *Suppose that*

$$\langle \sigma, T, e \rangle \longmapsto_w \langle \sigma', T', e' \rangle \longmapsto_w \langle \sigma'', T'', e'' \rangle$$

where the first transition is an instance of (Trans Activate)$_w$ or a protected transition and the second transition is an unprotected transition. Suppose further that $\langle \sigma, T, e \rangle$ is violation-free. Then there exists $\langle \sigma^*, T^*, e^* \rangle$ such that $\langle \sigma, T, e \rangle \longmapsto_w \langle \sigma^*, T^*, e^* \rangle \longmapsto_w \langle \sigma'', T'', e'' \rangle$ where the second transition is an instance of (Trans Activate)$_w$ or a protected transition.

PROOF. Consider a transition that is an instance of (Trans Activate)$_w$ followed by an unprotected transition:

$$\begin{array}{rl} \langle \sigma_k, e_1.\cdots.e_i.\cdots.e_n.e, \texttt{unit}\rangle & \longmapsto_w \langle \sigma_k, e_1.\cdots.e_i.\cdots.e_n, e\rangle \\ & \longmapsto_w \langle \sigma_{k+1}, e_1.\cdots.e'_i.\cdots.e_n, e\rangle \end{array}$$

(We place e at the end of $e_1.\cdots.e_i.\cdots.e_n.e$ only as a notational convenience; the argument is analogous in other cases.) We can rewrite this pair of transitions so that the unprotected transition is before the instance of (Trans Activate)$_w$:

$$\begin{array}{rl} \langle \sigma_k, e_1.\cdots.e_i.\cdots.e_n.e, \texttt{unit}\rangle & \longmapsto_w \langle \sigma_{k+1}, e_1.\cdots.e'_i.\cdots.e_n.e, \texttt{unit}\rangle \\ & \longmapsto_w \langle \sigma_{k+1}, e_1.\cdots.e'_i.\cdots.e_n, e\rangle \end{array}$$

The unprotected transition uses the same rule as before the commutation, with only a straightforward change in the choice of evaluation context that corresponds to the change from $e_1.\cdots.e_i.\cdots.e_n, e$ to $e_1.\cdots.e_i.\cdots.e_n.e, \texttt{unit}$.

Consider a protected transition followed by an unprotected transition:

$$\begin{array}{rl} \langle \sigma_k, e_1.\cdots.e_i.\cdots.e_n, e\rangle & \longmapsto_w \langle \sigma_{k+1}, e_1.\cdots.e_i.\cdots.e_n, e'\rangle \\ & \longmapsto_w \langle \sigma_{k+2}, e_1.\cdots.e'_i.\cdots.e_n, e'\rangle \end{array}$$

where the protected transition is not an instance of (Trans Async)$_w$. We can rewrite this pair of transitions so that the unprotected transition is before the protected transition:

$$\begin{array}{rl} \langle \sigma_k, e_1.\cdots.e_i.\cdots.e_n, e\rangle & \longmapsto_w \langle \sigma^*_{k+1}, e_1.\cdots.e'_i.\cdots.e_n, e\rangle \\ & \longmapsto_w \langle \sigma_{k+2}, e_1.\cdots.e'_i.\cdots.e_n, e'\rangle \end{array}$$

for an appropriate σ^*_{k+1}, which is constructed by case analysis on the protected transition:

—In the cases of (Trans Appl)$_w$ and (Trans Block)$_w$, we have that $\sigma_{k+1} = \sigma_k$, and we let $\sigma^*_{k+1} = \sigma_{k+2}$.

—In the case of (Trans Ref)$_w$, we have that $\sigma_{k+1} = \sigma_k[r \mapsto V]$ for some r and V. This location r is required not to be in $dom(\sigma_k)$, and hence (by our general requirements on states) cannot occur free in e_i. Therefore, the unprotected transition cannot read or write the contents of r, so $\sigma_{k+2}(r) = V$. Moreover, if the unprotected transition is itself an instance of (Trans Ref)$_w$, it cannot allocate the same r. We let σ^*_{k+1} be the restriction of σ_{k+2} to locations other than r. We have $\langle \sigma_k, e_1.\cdots.e_i.\cdots.e_n, e\rangle \longmapsto_w \langle \sigma^*_{k+1}, e_1.\cdots.e'_i.\cdots.e_n, e\rangle$ because the unprotected transition does not allocate, read, or write r. We have $\langle \sigma^*_{k+1}, e_1.\cdots.e'_i.\cdots.e_n, e\rangle \longmapsto_w \langle \sigma_{k+2}, e_1.\cdots.e'_i.\cdots.e_n, e'\rangle$ because the unprotected transition does not allocate or write r.

—In the case of (Trans Deref)$_w$, we have that $\sigma_{k+1} = \sigma_k$, and we let $\sigma^*_{k+1} = \sigma_{k+2}$. Thus, $\langle \sigma_k, e_1.\cdots.e_i.\cdots.e_n, e\rangle \longmapsto_w \langle \sigma^*_{k+1}, e_1.\cdots.e'_i.\cdots.e_n, e\rangle$. Because $\langle \sigma_k, e_1.\cdots.e_i.\cdots.e_n, e\rangle$ is violation-free, this unprotected transition cannot be a write to the location r read in (Trans Deref)$_w$, so $\sigma_{k+2}(r) = \sigma_k(r)$ and hence $\langle \sigma^*_{k+1}, e_1.\cdots.e'_i.\cdots.e_n, e\rangle \longmapsto_w \langle \sigma_{k+2}, e_1.\cdots.e'_i.\cdots.e_n, e'\rangle$.

—In the case of (Trans Set)$_w$, we have that $\sigma_{k+1} = \sigma_k[r \mapsto V]$ for some r and V. By our general requirements on states, we have that $r \in dom(\sigma_k)$. We let $\sigma^*_{k+1} =$

$\sigma_{k+2}[r \mapsto V_0]$ where $V_0 = \sigma_k(r)$. Because $r \in dom(\sigma_{k+1})$, the unprotected transition cannot allocate r, and because $\langle \sigma_k, e_1.\cdots.e_i.\cdots.e_n, e\rangle$ is violation-free, the unprotected transition cannot read or write r, so $\langle \sigma_k, e_1.\cdots.e_i.\cdots.e_n, e\rangle \longmapsto_w \langle \sigma^*_{k+1}, e_1.\cdots.e'_i.\cdots.e_n, e\rangle$. In addition, we obtain $\langle \sigma^*_{k+1}, e_1.\cdots.e'_i.\cdots.e_n, e\rangle \longmapsto_w \langle \sigma_{k+2}, e_1.\cdots.e'_i.\cdots.e_n, e'\rangle$ from $\sigma_{k+2} = \sigma^*_{k+1}[r \mapsto V]$ and by the definition of the transition relation in the case of (Trans Set)$_w$.

Similarly, consider a protected transition followed by an unprotected transition:

$$\begin{array}{rl} \langle \sigma_k, e_1.\cdots.e_i.\cdots.e_n, e\rangle & \longmapsto_w \ \langle \sigma_{k+1}, e''.e_1.\cdots.e_i.\cdots.e_n, e'\rangle \\ & \longmapsto_w \ \langle \sigma_{k+2}, e''.e_1.\cdots.e'_i.\cdots.e_n, e'\rangle \end{array}$$

where the protected transition is an instance of (Trans Async)$_w$, As in the cases of (Trans Appl)$_w$ and (Trans Block)$_w$, we have that $\sigma_{k+1} = \sigma_k$, and we let $\sigma^*_{k+1} = \sigma_{k+2}$. Note that we cannot have:

$$\begin{array}{rl} \langle \sigma_k, e_1.\cdots.e_n, e\rangle & \longmapsto_w \ \langle \sigma_{k+1}, e''.e_1.\cdots.e_n, e'\rangle \\ & \longmapsto_w \ \langle \sigma_{k+2}, e'''.e_1.\cdots.e_n, e'\rangle \end{array}$$

where the first transition is an instance of (Trans Async)$_w$ and the second transition operates immediately on the expression e'' added to the pool: this possibility is excluded by the restriction that in, an expression of the form `async` e'', any occurrences of `unprotected` are under a λ. □

Proof of Theorem 7.1. The proof is by induction on the length of $\langle \sigma, T, \texttt{unit}\rangle \longmapsto^*_w \langle \sigma', T', e'\rangle$. We strengthen the claim with the assertion that the strong computation is of the same length as the original computation.

The claim is vacuously true when this length is 0. For the inductive step, suppose that we have $\langle \sigma, T, \texttt{unit}\rangle \longmapsto^*_w \langle \sigma', T', e'\rangle \longmapsto_w \langle \sigma'', T'', e''\rangle$. By induction hypothesis, $\langle \sigma, T, \texttt{unit}\rangle \longmapsto^*_s \langle \sigma', T', e'\rangle \longmapsto_w \langle \sigma'', T'', e''\rangle$. If $\langle \sigma', T', e'\rangle \longmapsto_s \langle \sigma'', T'', e''\rangle$, then we immediately obtain a strong computation $\langle \sigma, T, \texttt{unit}\rangle \longmapsto^*_s \langle \sigma'', T'', e''\rangle$. On the other hand, if $\langle \sigma', T', e'\rangle \longmapsto_w \langle \sigma'', T'', e''\rangle$ is not strong, then there must be a transition before it (since otherwise $e' = e = \texttt{unit}$), in other words a last transition in $\langle \sigma, T, \texttt{unit}\rangle \longmapsto^*_s \langle \sigma', T', e'\rangle$. Moreover this strong transition to $\langle \sigma', T', e'\rangle$ cannot be an instance of (Trans Unprotect)$_w$ nor an unprotected strong transition, for otherwise $e' = \texttt{unit}$, so it must be an instance of (Trans Activate)$_w$ or a protected transition. By Lemma A.6, we can commute steps in order to obtain a strong computation $\langle \sigma, T, \texttt{unit}\rangle \longmapsto^*_s \langle \sigma'', T'', e''\rangle$. Specifically, Lemma A.6 shows how to move an unprotected transition before an instance of (Trans Activate)$_w$ or a protected transition. By induction hypothesis, the computation up to this unprotected transition can be made strong, and adding back the instance of (Trans Activate)$_w$ or the protected transition yields the desired strong computation $\langle \sigma, T, \texttt{unit}\rangle \longmapsto^*_s \langle \sigma'', T'', e''\rangle$.

Note that the new intermediate states generated in the proof could have violations, and the argument would collapse, if we had not assumed that all strong computations that start from the state $\langle \sigma, T, \texttt{unit}\rangle$ are violation-free. Considering strong computations is fortunately enough; considering only the given computation $\langle \sigma, T, \texttt{unit}\rangle \longmapsto^*_w \langle \sigma', T', e'\rangle$ would not be. □

A.4 Proofs for Section 8

In the weak semantics with roll-back, each transition has the form

$$\langle \sigma, T, e, f, l, P\rangle \longmapsto_{rw} \langle \sigma', T', e', f', l', P'\rangle$$

In many cases, a transition is defined in terms of a context that has a hole either in T and in T' or in e and in e'. Much as in Section 7, we say that the transition is protected if the hole is in e and in e', 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)$_{rw}$ or (Trans Block `false` Undo)$_{rw}$ are always protected;

—transitions that are instances of (Trans ... U)$_{rw}$ or of (Trans Close)$_{rw}$ are always unprotected;

—transitions that are instances of (Trans Unprotect)$_{rw}$, (Trans Block `false` Restore)$_{rw}$, (Trans Done)$_{rw}$, or (Trans Activate)$_{rw}$ are neither protected nor unprotected.

LEMMA A.7. *If* $\langle \sigma, T, \texttt{unit}, \texttt{unit}, \emptyset, \emptyset\rangle \longmapsto^*_{rs} \langle \sigma', T', e, f, l, P\rangle$*, where the first transition is an instance of (Trans Activate)*$_{rs}$ *and subsequent ones are not unprotected transitions, nor instances of (Trans Activate)*$_{rs}$ *or (Trans Block* `false` ... *)*$_{rs}$*, then* $\langle \sigma, T, \texttt{unit}\rangle \longmapsto^*_s \langle \sigma', T'.P, e\rangle$.

PROOF. The proof is by induction on the length of $\langle \sigma, T, \texttt{unit}, \texttt{unit}, \emptyset, \emptyset\rangle \longmapsto^*_{rs} \langle \sigma', T', e, f, l, P\rangle$. Applications of (Trans Done)$_{rs}$ in $\longmapsto_{rs}$ correspond to no transition with $\longmapsto_s$. In all other cases, each applicable transition rule for $\longmapsto^*_{rs}$ has a corresponding one for $\longmapsto^*_s$. □

LEMMA A.8. *If* $\langle \sigma, T, \texttt{unit}, \texttt{unit}, \emptyset, \emptyset\rangle \longmapsto^*_{rs} \langle \sigma', T', \texttt{unit}, \texttt{unit}, \emptyset, \emptyset\rangle$*, where the first transition is an instance of (Trans Activate)*$_{rs}$ *and subsequent ones are not unprotected transitions nor instances of (Trans Activate)*$_{rs}$*, and the computation includes instances of (Trans Block* `false` ... *)*$_{rs}$*, then* $T = T'$ *up to reordering and* σ' *extends* σ.

PROOF. The computation must end with a sequence of applications of (Trans Block `false` ...)$_{rs}$ that undo any updates to σ (possibly followed by a sequence of applications of (Trans Done)$_{rs}$ to $\langle \sigma', T', \texttt{unit}, \texttt{unit}, \emptyset, \emptyset\rangle$, with no effect). The first step takes an expression from T, while the application of Trans Block `false` Restore)$_{rs}$ returns it. □

LEMMA A.9. *If* $\langle \sigma, T, \texttt{unit}, \texttt{unit}, \emptyset, \emptyset\rangle \longmapsto^*_{rs} \langle \sigma', T', \texttt{unit}, \texttt{unit}, \emptyset, \emptyset\rangle$*, where the first transition is an instance of (Trans Activate)*$_{rs}$ *and subsequent ones are not unprotected transitions, nor instances of (Trans Activate)*$_{rs}$*, then there exist* σ'' *and* T'' *such that* σ' *extends* σ''*,* $T'' = T'$ *up to reordering, and* $\langle \sigma, T, \texttt{unit}\rangle \longmapsto^*_s \langle \sigma'', T'', \texttt{unit}\rangle$.

PROOF. Lemma A.7 deals with the case in which there are no occurrences of (Trans Block `false` ...)$_{rs}$ in the computation. Lemma A.8 deals with the case in which there are such occurrences. □

PROOF OF LEMMA 8.1. The proof is by induction on the length of

$$\langle \sigma, T, \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset \rangle \longmapsto_{rs}^{*} \langle \sigma', T', \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset \rangle$$

We distinguish three cases:

—Suppose that the computation consists of a single transition

$$\langle \sigma, T, \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset \rangle \longmapsto_{rs} \langle \sigma', T', \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset \rangle$$

This transition could be an unprotected transition, a trivial instance of (Trans Done)$_{rs}$, or a trivial instance of (Trans Activate)$_{rs}$. In all cases, we immediately have $\langle \sigma, T, \mathtt{unit} \rangle = \langle \sigma', T', \mathtt{unit} \rangle$ or $\langle \sigma, T, \mathtt{unit} \rangle \longmapsto_{s} \langle \sigma', T', \mathtt{unit} \rangle$.

—If there is any intermediate state of the form $\langle \sigma^*, T^*, \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset \rangle$, then we conclude by applying the induction hypothesis to

$$\langle \sigma, T, \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset \rangle \longmapsto_{rs}^{*} \langle \sigma^*, T^*, \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset \rangle$$

and to

$$\langle \sigma^*, T^*, \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset \rangle \longmapsto_{rs}^{*} \langle \sigma', T', \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset \rangle$$

—Otherwise, it must be that the first step in the computation is an instance of (Trans Activate)$_{rs}$, which is the only rule that can go from a state of the form $\langle \sigma'', T'', \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset \rangle$ to a state not of this same form. After this initial transition, unprotected transitions and (Trans Activate)$_{rs}$ are never enabled, because of the form of the intermediate states. We conclude by Lemma A.9.

□

PROOF OF LEMMA 8.3. We argue by contradiction. Suppose that a state with a violation was reachable: $\langle \sigma, T, \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset \rangle \longmapsto_{rs}^{*} \langle \sigma', T', e', f', l', P' \rangle$. We exclude any final occurrences of instances of (Trans Block `false` $\dots$)$_{rs}$ in this computation, since they cannot introduce any violations not already present. We show that $\langle \sigma, T, \mathtt{unit} \rangle \longmapsto_{s}^{*} \langle \sigma'', T'', e' \rangle$ where σ' extends σ'' and $T'' = T'.P'$ up to reordering. A violation in $\langle \sigma', T', e', f', l', P' \rangle$ immediately implies a violation in $\langle \sigma'', T'', e' \rangle$.

Lemma 8.1 enables us to simulate any prefixes

$$\langle \sigma, T, \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset \rangle \longmapsto_{rs}^{*} \langle \cdot, \cdot, \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset \rangle$$

of $\langle \sigma, T, \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset \rangle \longmapsto_{rs}^{*} \langle \sigma', T', e', f', l', P' \rangle$. After the longest such prefix, the first transition is an instance of (Trans Activate)$_{rs}$ and subsequent ones are not unprotected transitions, nor instances of (Trans Activate)$_{rs}$ or (Trans Block `false` $\dots$)$_{rs}$. Lemma A.7 yields that $\langle \sigma, T, \mathtt{unit} \rangle \longmapsto_{s}^{*} \langle \sigma'', T'', e' \rangle$ where σ' extends σ'' and $T'' = T'.P'$ up to reordering. □

LEMMA A.10. *Suppose that*

$$\langle \sigma, T, e, f, l, P \rangle \longmapsto_{rw} \langle \sigma', T', e', f', l', P' \rangle \longmapsto_{rw} \langle \sigma'', T'', e'', f'', l'', P'' \rangle$$

where the first transition is an instance of (Trans Activate)$_{rw}$ or a protected transition and the second transition is an unprotected transition. Suppose further that $\langle \sigma, T, e, f, l, P \rangle$ is violation-free, and that, if $r \in dom(l)$, then no element of T is of

the form $\mathcal{U}[\ !r\ ]$ *or* $\mathcal{U}[\ r := \ldots\ ]$. *Then there exists* $\langle \sigma^*, T^*, e^*, f^*, l^*, P^* \rangle$ *such that* $\langle \sigma, T, e, f, l, P \rangle \longmapsto_{rw} \langle \sigma^*, T^*, e^*, f^*, l^*, P^* \rangle \longmapsto_{rw} \langle \sigma'', T'', e'', f'', l'', P'' \rangle$ *where the second transition is an instance of (Trans Activate)*$_{rw}$ *or a protected transition.*

PROOF. The proof is analogous to that of Lemma A.6.

Consider a transition that is an instance of (Trans Activate)$_{rw}$ followed by an unprotected transition. As in Lemma A.6, we can rewrite this pair of transitions so that the unprotected transition is before the instance of (Trans Activate)$_{rw}$. The unprotected transition uses the same rule as before the commutation, with only a straightforward change in the choice of evaluation context.

Consider a protected transition followed by an unprotected transition. As in Lemma A.6, we can rewrite this pair of transitions so that the unprotected transition is before the protected transition, with an intermediate state constructed by case analysis on the protected transition. Most of the cases of protected transitions are as the corresponding cases in Lemma A.6.

—In the cases of (Trans Appl P)$_{rw}$, (Trans Block `true` P)$_{rw}$, (Trans Ref P)$_{rw}$, and (Trans Deref P)$_{rw}$, the state components f, l, and P are constant through the transitions, so we proceed exactly as in Lemma A.6, letting $f^* = f$, $l^* = l$, and $P^* = P$.

—The case of (Trans Async P)$_{rw}$ is similar except that P varies, and we let $P^* = P$.

—The case of (Trans Set P)$_{rw}$ is also similar except that l varies, and we let $l^* = l$.

—The case of (Trans Block `false` Undo)$_{rw}$ is analogous to that of (Trans Set P)$_{rw}$, and we let $l^* = l$ again. We use the hypothesis that if $r \in dom(l)$, then no element of T is of the form $\mathcal{U}[\ !r\ ]$ or $\mathcal{U}[\ r := \ldots\ ]$, instead of the hypothesis that the state $\langle \sigma, T, e, f, l, P \rangle$ is violation-free.

□

LEMMA A.11. *Assume that all strong computations with roll-back that start from the state* $\langle \sigma, T, \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset \rangle$ *are violation-free. If* $\langle \sigma, T, \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset \rangle \longmapsto^*_{rs} \langle \sigma', T', e', f', l', P' \rangle$ *then, if* $r \in dom(l')$, *then no element of* T' *is of the form* $\mathcal{U}[\ !r\ ]$ *or* $\mathcal{U}[\ r := \ldots\ ]$.

PROOF. We show that if $r \in dom(l')$, then no element of T' is of the form $\mathcal{U}[\ !r\ ]$ or $\mathcal{U}[\ r := \ldots\ ]$. We proceed by induction on the length of the computation, and argue by cases on the last step of the computation. If $l' = \emptyset$, then the claim is trivially true, so it suffices to consider the cases of rules (Trans $\ldots$ P)$_{rs}$ and (Trans Block `false` Undo)$_{rs}$. In all cases except that of (Trans Set P)$_{rs}$, we have that l' is equal or shorter than its previous value while T' equals its previous value, so the induction hypothesis yields the desired result. In the case of (Trans Set P)$_{rs}$, since the previous state is violation-free and its active expression is a write to the location r being added to l', the expressions in T' cannot be accessing this location. □

PROOF OF LEMMA 8.2. Consider a weak computation $\langle \sigma, T, \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset \rangle \longmapsto^*_{rw} \langle \sigma', T', e', f', l', P' \rangle$. Much as in Theorem 7.1, the proof is by induction on the length of this computation, relying on Lemma A.10. As in Theorem 7.1, we strengthen the claim with the assertion that the strong computation is of the same length as the original computation.

The claim is vacuously true when this length is 0. For the inductive step, suppose that we have

$$\begin{array}{rl} \langle \sigma, T, \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset \rangle & \longmapsto_{rw}^{*} \ \langle \sigma', T', e', f', l', P' \rangle \\ & \longmapsto_{rw} \ \langle \sigma'', T'', e'', f'', l'', P'' \rangle \end{array}$$

By induction hypothesis,

$$\begin{array}{rl} \langle \sigma, T, \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset \rangle & \longmapsto_{rs}^{*} \ \langle \sigma', T', e', f', l', P' \rangle \\ & \longmapsto_{rw} \ \langle \sigma'', T'', e'', f'', l'', P'' \rangle \end{array}$$

If $\langle \sigma', T', e', f', l', P' \rangle \longmapsto_{rs} \langle \sigma'', T'', e'', f'', l'', P'' \rangle$, then we immediately obtain a strong computation $\langle \sigma, T, \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset \rangle \longmapsto_{rs}^{*} \langle \sigma'', T'', e'', f'', l'', P'' \rangle$. On the other hand, if $\langle \sigma', T', e', f', l', P' \rangle \longmapsto_{rw} \langle \sigma'', T'', e'', f'', l'', P'' \rangle$ is not strong, then there must be a transition before it (since otherwise $e' = e = f' = f = \mathtt{unit}$ and $l' = l = P' = P = \emptyset$), in other words a last transition in $\langle \sigma, T, \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset \rangle \longmapsto_{rs}^{*} \langle \sigma', T', e', f', l', P' \rangle$. Moreover this strong transition to $\langle \sigma', T', e', f', l', P' \rangle$ cannot be an instance of (Trans Unprotect)$_{rs}$, (Trans Block `false` Restore)$_{rs}$, or (Trans Done)$_{rs}$ nor an unprotected strong transition, for otherwise $e' = f' = \mathtt{unit}$ and $l' = P' = \emptyset$), so it must be an instance of (Trans Activate)$_{rs}$ or a protected transition. We can apply Lemma A.10 because of Lemma A.11. By Lemma A.10, we can commute steps in order to obtain a strong computation $\langle \sigma, T, \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset \rangle \longmapsto_{rs}^{*} \langle \sigma'', T'', e'', f'', l'', P'' \rangle$. Specifically, Lemma A.10 shows how to move an unprotected transition before an instance of (Trans Activate)$_{rs}$ or a protected transition. By induction hypothesis, the computation up to this unprotected transition can be made strong, and adding back the instance of (Trans Activate)$_{rs}$ or the protected transition yields the desired strong computation $\langle \sigma, T, \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset \rangle \longmapsto_{rs}^{*} \langle \sigma'', T'', e'', f'', l'', P'' \rangle$. □

Proof of Theorem 8.4. The theorem follows from the composition of Lemmas 8.1 and 8.2. Lemma 8.3 implies that all strong computations with roll-back that start from the state $\langle \sigma, T, \mathtt{unit}, \mathtt{unit}, \emptyset, \emptyset \rangle$ are violation-free, so Lemma 8.2 applies. □

A.5 Proofs for Section 9

Lemmas A.1, A.2, and A.3 continue to hold, as they do not concern the operational semantics. The following are slight, trivial adaptations of Lemmas A.4 and A.5 that take into account the form of states used in this section.

Lemma A.12. *Assume that $r \in dom(\sigma)$ and $E(r) = \mathtt{Ref}_{p_0}\ t_0$. If $E \vdash \langle \sigma, T, O, l \rangle$ and $E\,; p_0 \vdash V : t_0$, then $E \vdash \langle \sigma[r \mapsto V], T, O, l \rangle$.*

Lemma A.13. *If $E \vdash \langle \sigma, T, O.(\mathcal{P}[\ e\ ], f, a, P).O', l \rangle$ then there exists t such that $E\,; \mathtt{P} \vdash e : t$. If $E \vdash \langle \sigma, T.\mathcal{U}[\ e'\ ].T', O, l \rangle$ then there exists t such that $E\,; \mathtt{U} \vdash e' : t$.*

Proof of Theorem 9.1. The proof is an extension of that of Theorem 6.1.

The cases of (Trans Appl P)$_{ow}$, (Trans Appl U)$_{ow}$, (Trans Ref P)$_{ow}$, (Trans Ref U)$_{ow}$, (Trans Deref P)$_{ow}$, (Trans Deref U)$_{ow}$, (Trans Set U)$_{ow}$, (Trans Async U)$_{ow}$, (Trans Block P)$_{ow}$, (Trans Block U)$_{ow}$, and (Trans Close)$_{ow}$ are like the corresponding cases in that proof.

In the case of (Trans Set P)$_{ow}$, we also have that the updated l is well-typed, and the location added to l has a type of the form $\mathtt{Ref}_{\mathtt{P}}\ t_0$ by Lemma A.13, which

gives the effect P to the assignment. In the case of (Trans Async P)$_{ow}$, similarly, we also have that the updated P is well-typed, straightforwardly.

The cases of (Trans Done)$_{ow}$ and (Trans Unprotect)$_{ow}$ are directly analogous to that of (Trans Unprotect)$_s$.

In the case of (Trans Undo)$_{ow}$, the updates to the store preserve typing by Lemma A.12, and the origin expressions (which are added back to the pool) are well-typed by hypothesis.

The case of (Trans Activate)$_{ow}$ is straightforward because the new active expression and its origin all come from the pool, which is well-typed by hypothesis, and l and P are empty in this case. □

Examining the proof again, we note 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 $\texttt{Ref}_{p_0}\ t_0$, we also note that if the transition is protected, then p_0 must be P, and if the transition is unprotected, then p_0 must be U, by Lemma A.13. It follows that there exist subsets P and U of $dom(\sigma')$ 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. As noted in the body of the paper, the subsets in question consist of the reference locations declared with effects P and U, respectively, in the environment.

Proof of Theorem 9.2. Assuming that $\langle \sigma, T\rangle$ is well-typed, more generally we consider a computation $\langle \sigma, T, \emptyset, \emptyset\rangle \longmapsto^*_{ow} \langle \sigma', T', O', l'\rangle$, and we prove that:

(1) 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'',
—$T'' = T'.origin(O')$ up to reordering.

(2) Moreover, if $O' = O^\dagger.(e, f, a, P).O^{\dagger\dagger}$ and (e, f, a, P) does not conflict with $O^\dagger.O^{\dagger\dagger}$, then there is a further strong computation

$$\begin{array}{rl} \langle \sigma'', T'', \texttt{unit}\rangle & \longmapsto_s \ \langle \sigma'', T''', f\rangle \\ & \longmapsto^*_s \ \langle \sigma''', T'''.P, e\rangle \end{array}$$

where the first transition is an instance of (Trans Activate)$_s$ and
—$\sigma'(l' - a)$ is an extension of σ''',
—$T''' = T'.origin(O^\dagger.O^{\dagger\dagger})$ up to reordering.

The proof is by induction on the computation $\langle \sigma, T, \emptyset, \emptyset\rangle \longmapsto^*_{ow} \langle \sigma', T', O', l'\rangle$, with a case analysis on the last rule applied. In most cases, the construction of a suitable strong computation $\langle \sigma, T, \texttt{unit}\rangle \longmapsto^*_s \langle \sigma'', T'', \texttt{unit}\rangle$ is straightforward using the induction hypothesis and typing. The cases of (Trans Unprotect)$_{ow}$ and (Trans Done)$_{ow}$ rely on the further strong computation of claim (2). Throughout, we work up to reorderings in the pool. We use the sets P and U as determined by $\langle \sigma, T, \emptyset, \emptyset\rangle \longmapsto^*_{ow} \langle \sigma', T', O', l'\rangle$ and Theorem 9.1.

—(Trans Appl P)$_{ow}$, (Trans Deref P)$_{ow}$, (Trans Async P)$_{ow}$, (Trans Block P)$_{ow}$:

(1) The desired strong computation is that given by the induction hypothesis, since $origin(O')$, σ', and l' do not change in these cases.
(2) The further strong computation is either that given by the induction hypothesis (if the transition belongs to a different try) or an extension by a corresponding application of $(\text{Trans Appl})_s$, $(\text{Trans Deref})_s$, $(\text{Trans Async})_s$, or $(\text{Trans Block})_s$, respectively (for the try where the transition operates).

—$(\text{Trans Ref P})_{ow}$:
(1) The desired strong computation is that given by the induction hypothesis, since $origin(O')$ does not change, σ' is extended with a new location, and l' does not change in this case.
(2) The further strong computation is either that given by the induction hypothesis (if the transition belongs to a different try) or an extension by a corresponding transition (for the try where the transition operates). In the latter case, we use that the reference location being allocated is not in the domain of $l' - a$.

—$(\text{Trans Set P})_{ow}$:
(1) The desired strong computation is that given by the induction hypothesis, since $origin(O')$ does not change, and since $\sigma' l'$ remains the same even if σ' and l' may change.
(2) The further strong computation is either that given by the induction hypothesis (if the transition belongs to a different try) or an extension by a corresponding transition (for the try where the transition operates). In the former case, the absence of conflict implies the required commutation: in case the instance of $(\text{Trans Set P})_{ow}$ adds a reference location in l', this reference location is not in a, so $\sigma'(l' - a) = \sigma^*(l^* - a)$, where σ^* and l^* are the reference store and the log before the transition. In the latter case, we use that $l' - a = l^* - a^*$, where l^* and a^* are the reference store and the list of accessed addresses of the try before the transition, and that the reference location being set is not in the domain of $l' - a$.

—$(\text{Trans Appl U})_{ow}$, $(\text{Trans Deref U})_{ow}$, $(\text{Trans Async U})_{ow}$, $(\text{Trans Block U})_{ow}$, $(\text{Trans Close})_{ow}$:
(1) The induction hypothesis gives a strong computation that we extend with a corresponding application of $(\text{Trans Appl})_s$, $(\text{Trans Deref})_s$, $(\text{Trans Async})_s$, $(\text{Trans Block})_s$, or $(\text{Trans Close})_s$, respectively.
(2) The further strong computation is that given by the induction hypothesis, with straightforward changes in the pool that correspond to the application of $(\text{Trans Appl})_s$, $(\text{Trans Deref})_s$, $(\text{Trans Async})_s$, $(\text{Trans Block})_s$, or $(\text{Trans Close})_s$, respectively.

—$(\text{Trans Ref U})_{ow}$:
(1) The induction hypothesis gives a strong computation that we extend with a corresponding application of $(\text{Trans Ref})_s$. Typing (and Theorem 9.1) imply that the application of the log l' and the allocation commute as they are to different locations, since the domain of a log consists of reference locations in `P`.
(2) The further strong computation is basically that given by the induction hypothesis, up to an extension in `U`, since a try can depend only on reference

locations in P, and typing implies that those cannot be allocated by (Trans Ref U)$_{ow}$.

—(Trans Set U)$_{ow}$:

(1) The induction hypothesis gives a strong computation that we extend with a corresponding application of (Trans Set)$_s$. Typing (and Theorem 9.1) imply that the application of the log l' and the update commute as they are to different locations, since the domain of a log consists of reference locations in P.

(2) The further strong computation is basically that given by the induction hypothesis, up to a change to the initial value of a reference location in U, since a try can depend only on reference locations in P, and typing implies that those cannot be updated by (Trans Set U)$_{ow}$.

—(Trans Activate)$_{ow}$:

(1) This case is also by a straightforward application of the induction hypothesis. The strong computation is that given by the induction hypothesis, since l' does not change, and since the origin of the new try comes from the pool.

(2) The further strong computation is either that given by the induction hypothesis (if the transition belongs to a different try) or an instance of (Trans Activate)$_s$ (for the try that the transition generates).

—(Trans Undo)$_{ow}$:

(1) Suppose that $\langle \sigma, T, \emptyset, \emptyset \rangle \longmapsto_{ow}^{*} \langle \sigma', T', O', l' \rangle$ where the last rule applied is an instance of (Trans Undo)$_{ow}$. If the state before the transition is $\langle \sigma^*, T^*, O^*, l^* \rangle$, then
—$\sigma' = \sigma^* l^*$,
—$T' = \mathit{origin}(O^*).T^*$,
—$O' = \emptyset$, and
—$l' = \emptyset$.
The induction hypothesis implies that 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. In other words, $\sigma' l'$ is an extension of σ'' and $T'' = T'.\mathit{origin}(O')$ up to reordering. So this strong computation satisfies the claim also for $\langle \sigma, T, \emptyset, \emptyset \rangle \longmapsto_{ow}^{*} \langle \sigma', T', O', l' \rangle$.

(2) This claim holds vacuously because O' is empty in this case.

—(Trans Unprotect)$_{ow}$:

(1) Suppose that $\langle \sigma, T, \emptyset, \emptyset \rangle \longmapsto_{ow}^{*} \langle \sigma', T', O', l' \rangle$ where the last rule applied is an instance of (Trans Unprotect)$_{ow}$. If the state before the transition is $\langle \sigma^*, T^*, O^*.(\mathcal{P}[\ \texttt{unprotected}\ e^*\ ], f^*, a^*, P^*).O^{**}, l^* \rangle$, then
—$\sigma^* = \sigma'$,
—$T' = T^*.\mathcal{P}[\ \texttt{unprotected}\ e^*\ ].P^*$,
—$O' = O^*.O^{**}$, and
—$l' = l^* - a^*$.
Moreover, $(\mathcal{P}[\ \texttt{unprotected}\ e^*\ ], f^*, a^*, P^*)$ does not conflict with $O^*.O^{**}$, that is, with O'.

The induction hypothesis implies that there is a strong computation

$$\langle \sigma, T, \texttt{unit} \rangle \longmapsto_s^* \langle \sigma'', T^*.origin(O^*.O^{**}).f^*, \texttt{unit} \rangle$$

where $\sigma' l^*$ is an extension of σ''. Moreover, the induction hypothesis implies that there is a further strong computation

$$\begin{array}{l} \langle \sigma'', T^*.origin(O^*.O^{**}).f^*, \texttt{unit} \rangle \\ \quad \longmapsto_s \ \langle \sigma'', T^*.origin(O^*.O^{**}), f^* \rangle \\ \quad \longmapsto_s^* \ \langle \sigma'', T^*.origin(O^*.O^{**}).P^*, \mathcal{P}[\ \texttt{unprotected}\ e^*\ ] \rangle \end{array}$$

where $\sigma'(l^* - a^*)$ is an extension of σ'', that is, where $\sigma' l'$ is an extension of σ''. We obtain the desired strong computation by applying $(\text{Trans Unprotect})_s$.

(2) The further strong computation is basically that given by the induction hypothesis, up to changes in the initial values of reference locations in a^*. The absence of conflict that is the hypothesis of the application of $(\text{Trans Unprotect})_{ow}$ implies that a and a^* do not intersect, so the initial values of reference locations in a^* do not affect a try that accesses only locations in a.

—$(\text{Trans Done})_{ow}$: This case is almost identical to that of $(\text{Trans Unprotect})_{ow}$ (but slightly simpler).

□

ACKNOWLEDGMENTS

We would like to thank Dan Grossman for interesting discussions on our work, Dave Detlefs for encouraging us to study the privatization problem, and Jean-Philippe Martin, Kate Moore, and Simon Peyton Jones for comments on earlier versions of this paper.

REFERENCES

Abadi, M. 2008. Automatic mutual exclusion and atomicity checks. In *Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, P. Degano, R. D. Nicola, and J. Meseguer, Eds. Lecture Notes in Computer Science, vol. 5065. Springer-Verlag, 510–526.

Abadi, M., Birrell, A., Harris, T., Hsieh, J., and Isard, M. 2008. Dynamic separation for transactional memory. Tech. Rep. MSR-TR-2008-43, Microsoft Research.

Abadi, M., Birrell, A., Harris, T., Hsieh, J., and Isard, M. 2009. Implementation and use of transactional memory with dynamic separation. In *CC '09: Proc. 18th International Conference on Compiler Construction*, O. de Moor and M. I. Schwartzbach, Eds. Lecture Notes in Computer Science, vol. 5501. Springer-Verlag, 63–77.

Abadi, M., Flanagan, C., and Freund, S. N. 2006. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst. 28,* 2, 207–255.

Abadi, M., Harris, T., and Moore, K. F. 2008. A model of dynamic separation for transactional memory. In *CONCUR '08: Proc. 19th International Confernece on Concurrency Theory*, F. van Breugel and M. Chechik, Eds. Lecture Notes in Computer Science, vol. 5201. Springer-Verlag, 6–20.

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

Adve, S. V. and Hill, M. D. 1990. Weak ordering – a new definition. *SIGARCH Comput. Archit. News 18,* 3a, 2–14.

Adya, A., Howell, J., Theimer, M., Bolosky, W. J., and Douceur, J. R. 2002. Cooperative task management without manual stack management. In *USENIX '02: Proc. 2002 USENIX Annual Technical Conference*. 289–302.

Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., Steele Jr., G. L., and Tobin-Hochstadt, S. 2007. The Fortress language specification, v1.0β. Tech. rep., Sun Microsystems.

Blundell, C., Lewis, E. C., and Martin, M. M. K. 2006. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters 5,* 2.

Bronson, N. G., Kozyrakis, C., and Olukotun, K. 2009. Feedback-directed barrier optimization in a strongly isolated STM. In *POPL '09: Proc. 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 213–225.

Carlstrom, B. D., McDonald, A., Chafi, H., Chung, J., Minh, C. C., Kozyrakis, C., and Olukotun, K. 2006. The Atomos transactional programming language. In *PLDI '06: Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1–13.

Cohen, A., O'Leary, J. W., Pnueli, A., Tuttle, M. R., and Zuck, L. D. 2007. Verifying correctness of transactional memories. In *FMCAD '07: Proc. 7th International Conference on Formal Methods in Computer-Aided Design*. 37–44.

Dalessandro, L. and Scott, M. L. 2009. Strong isolation is a weak idea. In *TRANSACT '09: 4th Workshop on Transactional Computing*.

Dice, D., Shalev, O., and Shavit, N. 2006. Transactional locking II. In *DISC '06: Proc. 20th International Symposium on Distributed Computing*, S. Dolev, Ed. Lecture Notes in Computer Science, vol. 4167. Springer, 194–208.

Dice, D. and Shavit, N. 2006. What really makes transactions faster? In *TRANSACT '06: Proc. ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. http://hdl.handle.net/1802/4051.

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

Guerraoui, R., Henzinger, T., and Singh, V. 2008a. Model checking transactional memories. In *PLDI '08: Proc. 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 372–382.

Guerraoui, R., Henzinger, T. A., and Singh, V. 2008b. Completeness and nondeterminism in model checking transactional memories. In *CONCUR '08: Proc. 19th International Confernece on Concurrency Theory*, F. van Breugel and M. Chechik, Eds. Lecture Notes in Computer Science, vol. 5201. Springer-Verlag, 21–35.

Harris, T. and Fraser, K. 2003. Language support for lightweight transactions. In *OOPSLA '03: Proc. 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 388–402.

Harris, T., Marlow, S., Peyton Jones, S., and Herlihy, M. 2005. Composable memory transactions. In *PPoPP '05: Proc. 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 48–60.

Harris, T., Plesko, M., Shinnar, A., and Tarditi, D. 2006. Optimizing memory transactions. In *PLDI '06: Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 14–25.

Igarashi, A., Pierce, B. C., and Wadler, P. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst. 23,* 3, 396–450.

Isard, M. and Birrell, A. 2007. Automatic mutual exclusion. In *HotOS '07: Proc. 11th Workshop on Hot Topics in Operating Systems*.

Jagannathan, S., Vitek, J., Welc, A., and Hosking, A. 2005. A transactional object calculus. *Science of Computer Programming 57,* 2, 164–186.

Kuszmaul, B. C. and Leiserson, C. E. 2003. Transactions everywhere. Tech. rep., MIT. http://hdl.handle.net/1721.1/3692.

Liblit, B. 2006. An operational semantics for LogTM. Tech. Rep. 1571, U. Wisconsin–Madison. Version 1.0.

Manson, J., Baker, J., Cunei, A., Jagannathan, S., Prochazka, M., Xin, B., and Vitek, J. 2005. Preemptible atomic regions for real-time Java. In *RTSS '05: Proc. 26th IEEE Real-Time Systems Symposium*. 62–71.

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

Menon, V., Balensiefer, S., Shpeisman, T., Adl-Tabatabai, A.-R., Hudson, R., Saha, B., and Welc, A. 2008. Practical weak-atomicity semantics for Java STM. In *SPAA '08: Proc. 20th Annual Symposium on Parallelism in Algorithms and Architectures*. 314–325.

Moore, K. E., Bobba, J., Moravan, M. J., Hill, M. D., and Wood, D. A. 2006. LogTM: Log-based transactional memory. In *HPCA '06: Proc. 12th International Symposium on High-Performance Computer Architecture*. 254–265.

Moore, K. F. and Grossman, D. 2008. High-level small-step operational semantics for transactions. In *POPL '08: Proc. 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 51–62.

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

Saha, B., Adl-Tabatabai, A.-R., Hudson, R. L., Minh, C. C., and Hertzberg, B. 2006. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proc. 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 187–197.

Schneider, F. T., Menon, V., Shpeisman, T., and Adl-Tabatabai, A.-R. 2008. Dynamic optimization for efficient strong atomicity. In *OOPSLA '08: Proc. 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*. 181–194.

Scott, M. L. 2006. Sequential specification of transactional memory semantics. In *TRANSACT '06: Proc. 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. http://hdl.handle.net/1802/4050.

Shavit, N. and Touitou, D. 1995. Software transactional memory. In *PODC '95: Proc. 14th ACM Symposium on Principles of Distributed Computing*. 204–213.

Shpeisman, T., Menon, V., Adl-Tabatabai, A.-R., Balensiefer, S., Grossman, D., Hudson, R. L., Moore, K. F., and Saha, B. 2007. Enforcing isolation and ordering in STM. In *PLDI '07: Proc. 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 78–88.

Spear, M. F., Dalessandro, L., Marathe, V., and Scott, M. L. 2008. Ordering-based semantics for software transactional memory. In *OPODIS '08: Proc. 12th International Conference on Principles of Distributed Systems*, T. P. Baker, A. Bui, and S. Tixeuil, Eds. Lecture Notes in Computer Science, vol. 972. Springer, 275–294.

Spear, M. F., Marathe, V. J., Dalessandro, L., and Scott, M. L. 2007. Privatization techniques for software transactional memory. Tech. Rep. 915, University of Rochester.

Sterling, N. 1993. Warlock: A static data race analysis tool. In *Proc. USENIX Winter Technical Conference*. 97–106.

Tasiran, S. 2008. A compositional method for verifying software transactional memory implementations. Tech. Rep. MSR-TR-2008-56, Microsoft Research.

Wang, C., Chen, W.-Y., Wu, Y., Saha, B., and Adl-Tabatabai, A.-R. 2007. Code generation and optimization for transactional memory constructs in an unmanaged language. In *CGO '07: Proc. International Symposium on Code Generation and Optimization*. 34–48.

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

Wright, A. K. and Felleisen, M. 1994. A syntactic approach to type soundness. *Information and Computation 115,* 1, 38–94.

Received November 2008; . . . ; accepted . . .