Automatic Mutual Exclusion and Atomicity Checks

Martín Abadi[1,2]

[1] Microsoft Research
[2] University of California, Santa Cruz

Abstract. This paper provides an introduction to the Automatic Mutual Exclusion (AME) programming model and to its formal study, through the AME calculus. AME resembles cooperative multithreading; in the intended implementations, however, software transactional memory supports the concurrent execution of atomic fragments. This paper also studies simple dynamic and static mechanisms for atomicity checks in AME.

1 Introduction

Transactions promise a practical mechanism for synchronization that should facilitate the design and coding of a wide range of concurrent systems. In particular, in shared-memory concurrency, systems based on transactions may achieve the efficiency of fine-grained locking while reducing the opportunities for deadlocks, race conditions, and other bugs. For these benefits to be realized, however, advances in low-level implementations of transactions do not suffice. Also needed are corresponding languages and programming techniques (e.g., [9, 10, 6, 3]).

The principle "Lo bueno, si breve, dos veces bueno" does not necessarily apply to transactions. Although long-running transactions can lead to excessive conflicts and may complicate hardware-based implementation strategies, they also support a conservative style of programming in which transactions, with their guarantees, are the default. This style is embodied, in particular, in the Automatic Mutual Exclusion (AME) model [12, 1], which is the focus of this paper.

AME can be seen as cooperative multithreading on top of software transactional memory (STM) [14]. In the spirit of cooperative multithreading, calls to the construct `yield` delimit atomic fragments of computations. STM allows multiple sequential code fragments to execute at the same time, each within a transaction.

Yielding requires care. For instance, consider a call to a library method made from within a transaction. As long as the execution remains within the same transaction, the caller need not be concerned with concurrent calls to the library or any other concurrent activity. On the other hand, the library method may decide to interrupt the transaction by yielding, perhaps in order to interact with the outside world. In this case, the caller may need to consider interleavings of

other computations, restoring invariants if necessary. In this paper we explore a mechanism for asserting that, dynamically, yielding should not happen in a particular piece of code. Yielding can be turned into (caught) run-time errors, and transactional recovery may optionally mask those errors altogether. We also define and study a simple static type system that indicates whether yielding is possible in a piece of code. A practical version of this type system has been implemented for an extension of C# on Bartok-STM [11].

In sum, the goals of this paper are to provide an introduction to AME and to its formal study (largely as a review of recent work [12, 1]), and also to advance a specific aspect of AME and its theory. Section 2 describes AME, informally. Sections 3 and 4 define the AME calculus and its high-level formal semantics. Section 5 and 6 concern dynamic atomicity checks and the static type system, respectively. Section 7 establishes the soundness of the static type system with respect to the dynamic atomicity checks. Section 8 concludes by mentioning some further work. An appendix contains proofs.

Similar themes have been explored in other projects. For instance, in the Mianjin language, type annotations distinguish routines that may perform communication [13]. More recently (independently from the AME work), the model Transactions with Isolation and Cooperation (TIC) includes a type system for atomicity [15]. In both Mianjin and TIC, the type systems are defined semi-formally. Further, other research on types for atomicity offers powerful analyses that apply to Java and similar languages [7]. While some of their ideas may be useful in implementations of AME, they may be less necessary at the AME source level, because of the reliance on cooperation and transactions. In another direction, research on sagas explores techniques that reconcile atomicity and responsiveness for long-lived transactions, with sophisticated treatments of nesting, parallelism, and compensation (which are beyond the scope of the present paper) [8, 5]. Finally, research on cooperative multithreading includes techniques for proving that yielding must eventually happen, guaranteeing fairness in single-threaded implementations [4].

2 Automatic Mutual Exclusion

AME encourages programmers to use transactions: code is executed in transactions by default. The intent is that the pervasive use of transactions will lead to clearer programs with fewer synchronization bugs. However, for interactions with legacy components and other computations that should not be placed in transactions, code can be marked explicitly as "unprotected".

In AME, running a program consists of executing a set of asynchronous method calls. The semantics of AME guarantees that the program execution is equivalent to executing each of these calls (or their fragments, as explained below) in some serial order. An asynchronous method call is created by an invocation `async MethodName(<args>)`. The caller continues immediately after this invocation. AME achieves concurrency by executing asynchronous method calls in transactions, overlapping the execution of multiple calls, with roll-backs

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

Fig. 1. Syntax of the AME calculus (without unprotected sections).

when conflicts occur. If a transaction initiates other asynchronous method calls, their execution is deferred until the initiating transaction commits, and they are discarded if the initiating transaction aborts.

An asynchronous method call may also invoke `yield`. A `yield` call breaks a method into multiple atomic fragments, implemented by committing one transaction and starting a new one. These atomic fragments are delimited dynamically by the calls to `yield`, not statically scoped like explicit atomic blocks [7, 9]. AME thus avoids some of the pitfalls of pure event-based programming models (in particular, "stack ripping" [2].). With this addition, the overall execution of a program is guaranteed to be a serialization of its atomic fragments.

An atomic fragment may include any number of guards, each of the form `blockUntil(<predicate>)`. An atomic fragment executes to completion only if all the guards encountered in the course of the execution have predicates that evaluate to true. The implementation of `blockUntil` does nothing if the predicate holds, but otherwise it aborts the current atomic fragment and re-executes it later (at a time when it is likely to succeed).

As indicated above, AME provides block-structured `unprotected` sections. We omit them here, for simplicity. It is straightforward to extend the results of this paper to them, although the semantics of `unprotected` sections can be delicate.

3 The AME Calculus

In our formal study of AME, we focus on a small but expressive language that we call the AME calculus. This calculus includes constructs for AME, higher-order functions, and imperative features.

In Figure 1 we define the syntax of the AME calculus, omitting unprotected sections. This syntax is untyped; we define a type system in Section 6. 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 `yield`, 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).

We write `let` $x = e$ `in` e' for $(\lambda x.\, e')\, e$, and also write $e; e'$ for `let` $x = e$ `in` e' when x does not occur free in e'. Including standard control structures and other common constructs (directly or by encodings) is routine.

As a small example, let us consider the following code fragment:

```
blockUntil !r0;
r1 := e1;
r2 := e2;
async (r3 := e3);
yield
```

in which r_0, r_1, r_2, and r_3 are variables that represent reference locations, and e_1, e_2, and e_3 are arbitrary expressions. This code fragment blocks until r_0 holds `true`, then it performs assignments to r_1 and r_2, forks an expression that will perform an assignment to r_3, and finally yields.

Intuitively, a programmer may expect that the assignments to r_1 and r_2 (but not r_3) happen within the same transaction, and this property will indeed hold if e_1 and e_2 are simple values. However, in general, the evaluations of e_1 and e_2 may trigger calls to `yield`, so the assignments may happen in different transactions. For instance, e_2 might be a call to a function with body `yield`; (`blockUntil` $!r_4$); $!r_5$, which yields, waits until the value in r_4 is true, and then returns the value in r_5. In that case, some other thread may execute between the assignments, may observe inconsistent values in r_1 and r_2, and may misbehave as a result. Therefore, it is useful to have dynamic or static means of guaranteeing that expressions such as e_1 and e_2 do not yield. Sections 5 and 6 address this goal.

4 High-Level Semantics

This section presents a semantics for the AME calculus. This semantics is intended to provide a clear, high-level model, rather than a description of possible underlying implementation techniques. Accordingly, the semantics does not model optimistic concurrency, conflict detection, roll-back, and other important low-level features. In [1] we consider richer and weaker semantics that add these features. Those weaker semantics implement the high-level semantics—though under non-trivial assumptions that restrict the sharing of data between transactions and unprotected code.

4.1 States

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

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

Fig. 2. State space.

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

A reference store σ is a finite mapping of reference locations to values. Formally, reference locations are 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.

4.2 Steps

A transition relation takes an execution from one state to the next. According to this transition relation, when the active expression is `unit`, an expression from the pool becomes the active expression. It is then evaluated as such until it produces `unit` or until it yields. No other computation is interleaved with this evaluation. Each evaluation step produces a new state. Unless the active expression is `unit`, this new state is obtained by decomposing the active expression into an evaluation context and a subexpression that describes an operation (for instance, a function application or an allocation).

As usual, a context is an expression with a hole [ ], and an evaluation context is a context of a particular kind. Given a context $\mathcal{C}$ and an expression e, we write $\mathcal{C}[\ e\ ]$ for the result of placing e in the hole in $\mathcal{C}$. We use the evaluation contexts defined by the grammar:

$$\mathcal{P} = [\ ] \mid \mathcal{P}\ e \mid V\ \mathcal{P} \mid \texttt{ref}\ \mathcal{P} \mid\ !\mathcal{P} \mid \mathcal{P} := e \mid r := \mathcal{P} \mid \texttt{blockUntil}\ \mathcal{P}$$

Figure 3 gives rules that specify the transition relation. The string "Trans" in the names of the rules refers to "transition" rules, not to "transaction". In these rules, we write $e[V/x]$ for the result of the capture-free substitution of V for x in e, and write $\sigma[r \mapsto V]$ for the store that agrees with σ except at r, which is mapped to V.

(Trans Activate) applies when the active expression is `unit` and the pool is not empty; it takes an expression from the pool as the new active expression. In all other rules, a subexpression in an evaluation context in the active expression determines a possible next operation. For instance, in (Trans Appl), the

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

Fig. 3. Transition rules of the abstract machine.

subexpression is a function application $(\lambda x.\ e)\ V$, so the next operation is beta reduction, and the result $e[V/x]$ of this beta reduction replaces $(\lambda x.\ e)\ V$ in the evaluation context. Similarly, in (Trans Yield), the subexpression is `yield`, so `unit` replaces `yield`, the active expression is moved to the pool, and the new active expression is `unit`. No rule applies in some cases, for instance when the active expression is `blockUntil false`. Lower-level semantics may abort and roll-back in such cases [1].

These rules are more compact than previous ones, simply because of the omission of unprotected computations. Further variants are possible. In particular, we may consider adding the rule:

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

This rule represents a short-cut: it can be derived by composing (Trans Yield) and (Trans Activate).

5 Dynamic Atomicity Checks

We extend the calculus with a construct that asserts the absence of yielding in a computation. We focus on the high-level semantics of Section 4, though similar extensions and corresponding results can be obtained for other semantics.

The extension goes as follows:

– We extend the syntax of the language with terms of the form $\langle e \rangle$. Informally, $\langle e \rangle$ means that there should be no yield in the course of the evaluation of e.

(This notation is inspired by Lamport's angle brackets, which also indicate atomicity.)

– We also extend the evaluation contexts, so that evaluation can proceed under $\langle\, \cdot\, \rangle$. Their grammar becomes:

$$\mathcal{P} = [\,] \mid \mathcal{P}\ e \mid V\ \mathcal{P} \mid \texttt{ref}\ \mathcal{P} \mid\ !\mathcal{P} \mid \mathcal{P} := e \mid r := \mathcal{P} \mid \texttt{blockUntil}\ \mathcal{P} \mid \langle \mathcal{P} \rangle$$

– We extend all the rules of the operational semantics to these terms and these evaluation contexts, and also add a rule to the operational semantics:

$$\langle \sigma, T, \mathcal{P}[\ \langle V \rangle\ ] \rangle \longmapsto \langle \sigma, T, \mathcal{P}[\ V\ ] \rangle \quad \text{(Trans Assert)}$$

Given that $\langle e \rangle$ asserts that there is no yield in the course of the evaluation of e, this rule says that the assertion can be dismissed when e is a value V (not subject to further evaluation).

These extensions are conservative, in the sense that they affect neither the operational semantics nor the typing (in the type system of Section 6) of expressions without assertions. Therefore, some of the main results below (Theorems 1 and 3) apply also without the extensions.

Consider a transition that is an instance of (Trans Yield), so this transition is of the form:

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

for some σ, T, and $\mathcal{P}$. We say that this transition is an atomicity violation if $\mathcal{P}$ is of the form $\mathcal{P}'[\ \langle \mathcal{P}'' \rangle\ ]$, for instance if $\mathcal{P}[\ \texttt{yield}\ ]$ is $\langle \texttt{yield} \rangle$ or $!\langle \texttt{ref yield} \rangle$.

What should we do with an atomicity violation? There are at least three distinct possibilities:

1. Continue the computation despite the atomicity violation; in this case, the main use of $\langle\, \cdot\, \rangle$ is as a marker that allows us to explain what went wrong. The present definition of the operational semantics embodies this possibility. Accordingly, the results below concern this possibility as well.
2. Stop the computation, allowing for recovery.
   Formally, it would suffice to remove the transitions that constitute atomicity violations, with the understanding that any computation that has not committed may be rolled back, and perhaps retried later. Specifically, we would restrict (Trans Yield) to:

$$\langle \sigma, T, \mathcal{P}[\ \texttt{yield}\ ] \rangle \longmapsto \langle \sigma, T.\mathcal{P}[\ \texttt{unit}\ ], \texttt{unit} \rangle \quad \text{if } \mathcal{P} \text{ is not of the form } \mathcal{P}'[\ \langle \mathcal{P}'' \rangle\ ]$$

   Thus, $\langle \mathcal{P}''[\texttt{yield}] \rangle$ would be analogous to `blockUntil false`.
3. Stop the computation with a fatal error.
   Formally, we could add a special state `wrong` that would represent errors, and change the operational semantics for producing errors instead of allowing atomicity violations. Specifically, we would restrict (Trans Yield), as above, and add:

$$\langle \sigma, T, \mathcal{P}'[\ \langle \mathcal{P}''[\texttt{yield}] \rangle\ ] \rangle \longmapsto \texttt{wrong}$$

$$
\begin{array}{rl}
s, t \in \mathit{Type} & = \texttt{Unit} \\
& \mid \texttt{Bool} \\
& \mid s \to^{p} t \\
& \mid \texttt{Ref}\ t \\
p, q \in & \{\texttt{Yields}, \texttt{NoYields}\}
\end{array}
$$

Fig. 4. Types for yielding.

With all these options, it is attractive to prove that, for some class of good programs, atomicity violations are not possible. The next section provides a type system for this purpose.

6 Static Atomicity Checks

This section defines a simple type system for atomicity checking. This type system can be seen as an alternative to the dynamic approach described above in Section 5. However, the two approaches may be combined; moreover, the dynamic approach is useful for formulating the correctness of the static approach (in Section 7).

The type system is based on the syntax of types of Figure 4, and is defined in terms of formal 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 p

The typing rules of Figure 5 operate on these judgments.

The type of an expression depends on a typing environment E, which maps variables to types. The typing environment is organized as a sequence of bindings, and we use $\emptyset$ to denote the empty environment:

$$E ::= \emptyset \mid E, x : t$$

The core of the type system is the set of rules for the judgment $E\,; p \vdash e : t$ (read "e is a well-typed expression of type t in typing environment E with effect p"). The intent is that, if this judgment holds, then e yields values of type t with effect p, and the free variables of e are given bindings consistent with the typing environment E. When p is `Yields`, this means that the evaluation of e may yield; when p is `NoYields`, this means that the evaluation of e definitely does not yield. We require that $\langle\, \cdot\, \rangle$ appears only around expressions with effect `NoYields`. We write $q <: p$ for $p = q$ or $p = \texttt{Yields}$. We say that e is well-typed when there exist E, p, and t such that $E\,; p \vdash e : t$.

As a design choice, we arrange that every expression that can be typed with effect `NoYields` can also be typed with effect `Yields`. For instance, we allow giving the effect `Yields` to the constant `true`, although the evaluation of `true` will

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{E \vdash \diamond}{E\,;\texttt{Yields} \vdash \texttt{yield} : \texttt{Unit}} \qquad \text{(Exp Yield)}$$

$$\frac{E\,;\texttt{NoYields} \vdash e : t}{E\,;p \vdash \langle e \rangle : t} \qquad \text{(Exp Assert)}$$

Fig. 5. Rules of the first-order type system for yielding.

obviously never yield. This property ensures that effects are not invalidated by computation. For example, consider the expression `yield`; `true`, which has effect `Yields` and produces the result `true`. Because `true` has effects `NoYields` and also `Yields`, the effect of `yield`; `true` continues to be derivable after reduction to `true`.

There are alternative methods for achieving the same effect. These include the use of a system with subtyping, which would also provide more flexibility at function types. The present method is simpler and enables us to focus on the core system. Undoubtedly richer type disciplines are possible.

7 Soundness

Intuitively, the correctness of the type system is the property that says that if an expression has effect `NoYields` statically then it does not yield at run-time. However, in the course of evaluation, the expression may change, and that should not be an excuse for yielding. So it is convenient to tag the expression, and to keep the tag on the expression even if the expression changes until its evaluation completes. The angle brackets of Section 5 serve as such a tag.

As a first step in the soundness proof, we generalize the type system to states $\langle \sigma, T, e \rangle$. We write

$$E\,; p_1 . \cdots . p_n, p \vdash \langle \sigma, e_1 . \cdots . e_n, e \rangle$$

if

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

The first condition relates the domains of σ and E. The second one says that E assigns types of the appropriate form to reference locations, and that σ maps these reference locations to expressions of appropriate types, with effect `NoYields` (because these expressions must be values). The remaining conditions require typing the expressions e_1, ..., e_n, and e.

We say that $\langle \sigma, e_1 . \cdots . e_n, e \rangle$ is well-typed if there exist E and p_1, ..., p_n, p such that $E\,; p_1 . \cdots . p_n, p \vdash \langle \sigma, e_1 . \cdots . e_n, e \rangle$.

We obtain that typability is preserved by computation:

Theorem 1 (Preservation of Typability). *If $\langle \sigma, T, e \rangle \longmapsto^* \langle \sigma', T', e' \rangle$ and $\langle \sigma, T, e \rangle$ is well-typed, then so is $\langle \sigma', T', e' \rangle$.*

Partly as a corollary, we also obtain a result that expresses the correctness of `NoYields`:

Theorem 2 (Atomicity Soundness). *If $\langle \sigma, T, e \rangle \longmapsto^* \langle \sigma', T', e' \rangle$ and $\langle \sigma, T, e \rangle$ is well-typed, then none of the transitions in $\langle \sigma, T, e \rangle \longmapsto^* \langle \sigma', T', e' \rangle$ is an atomicity violation.*

Moreover, we obtain a progress result, which characterizes when a computation may stop and implies that computations do not get stuck in unexpected ways:

Theorem 3 (Progress). *If $\langle \sigma, T, e \rangle$ is well-typed, the only free variables in $\langle \sigma, T, e \rangle$ are reference locations, and $\langle \sigma, T, e \rangle \longmapsto^* \langle \sigma', T', e' \rangle$, then:*

1. *there exists $\langle \sigma'', T'', e'' \rangle$ such that $\langle \sigma', T', e' \rangle \longmapsto \langle \sigma'', T'', e'' \rangle$; or*
2. *e' is of the form $\mathcal{P}[$ `blockUntil false` $]$; or*
3. *e' is* `unit` *and T' is empty.*

The proofs of these three theorems are in an appendix.

8 Further Work

This paper provides an introduction to the AME programming model and advances one aspect of its development and formal study. We conclude with a brief description of other recent and ongoing work on this model.

To date, we have only limited experience in programming in the AME model. While this experience is rather encouraging, further experience may conceivably lead to refinements in the constructs for AME. For instance, we have briefly considered expressive generalizations of `yield`. In any case, it seems likely that the need for atomicity checking will persist.

The semantics presented in this paper is a high-level description of the intended meanings of the AME constructs. Lower-level semantics embody various strategies for the implementation of these constructs. For instance, those lower-level semantics can include optimistic concurrent execution of transactions, with in-place updates to memory, conflict detection, and roll-backs [11]. In particular, the implementation of AME for C# on Bartok-STM relies on these features. Such strategies may have great advantages in performance and responsiveness, but they can lead to surprising results. We have therefore worked on describing those strategies precisely and on analyzing their properties in detail [1]. The correctness of these strategies require substantial assumptions which say, roughly that transactional and non-transactional computations do not share data directly. Several versions of these assumptions lead to correctness results, though with different specifics. Some of these versions, and the corresponding trade-offs, are the subject of ongoing work.

Acknowledgements

This paper is based on the original work on AME by Michael Isard and Andrew Birrell, and on further, ongoing joint work with Tim Harris and Johnson Hsieh. Dan Grossman made useful comments on the type system of Section 6. I am grateful to all of them.

References

1. Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *Proc. 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 63–74, 2008.
2. Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *Proc. 2002 USENIX Annual Technical Conference*, pages 289–302, 2002.
3. Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress language specification, v1.0β. Technical report, Sun Microsystems, March 2007.
4. Gérard Boudol. Fair cooperative multithreading. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR 2007 - Concurrency Theory, 18th International Conference*, volume 4703 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2007.
5. Roberto Bruni, Hernán C. Melgratti, and Ugo Montanari. Theoretical foundations for compensations in flow composition languages. In *Proc. 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 209–220, 2005.
6. Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The Atomos transactional programming language. In *PLDI '06: Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–13, 2006.
7. Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proc. 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 338–349, 2003.
8. Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proc. ACM SIGMOD 1987 Annual Conference*, pages 249–259, 1987.
9. Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proc. 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, 2003.
10. Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proc. 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60, 2005.
11. Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 14–25, 2006.
12. Michael Isard and Andrew Birrell. Automatic mutual exclusion. In *Proc. 11th Workshop on Hot Topics in Operating Systems*, 2007.
13. Paul Roe and Clemens A. Szyperski. Mianjin: A parallel language with a type system that governs global system behaviour. In Jürg Gutknecht and Wolfgang Weck, editors, *Modular Programming Languages, Joint Modular Languages Conference, JMLC 2000*, volume 1897 of *Lecture Notes in Computer Science*, pages 38–50. Springer, 2000.
14. Nir Shavit and Dan Touitou. Software transactional memory. In *Proc. 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, 1995.
15. Yannis Smaragdakis, Anthony Kay, Reimer Behrends, and Michal Young. Transactions with isolation and cooperation. In *Proc. 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 191–210, 2007.

16. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

Appendix: Proofs

Auxiliary Results. We rely on a few auxiliary results. Several of them are routine, and we omit the corresponding proofs. These include a replacement lemma (in the style of Wright and Felleisen [16]), a substitution lemma, and a lemma that deals with updates to the state.

Lemma 1 (Replacement). *Consider a derivation $\mathcal{D}$ of $E\,;p \vdash \mathcal{P}[\ e_0\ ] : t$. 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 $\mathcal{P}[\ \cdot\ ]$. 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 $\mathcal{P}$. Then $\mathcal{D}'$ is a derivation of $E\,;p \vdash \mathcal{P}[\ e'_0\ ] : t$.*

Lemma 2 (Substitution). *If $E, x : s, E'\,;p \vdash e : t$ and $E\,;\mathtt{NoYields} \vdash e' : s$ then $E, E'\,;p \vdash e[e'/x] : t$.*

Lemma 3 (Update). *Assume that $r \in dom(\sigma)$ and $E(r) = \mathtt{Ref}\ t_0$. If $E\,;p_1.\cdots.p_n, p \vdash \langle \sigma, e_1.\cdots.e_n, e\rangle$ and $E\,;\mathtt{NoYields} \vdash V : t_0$, then $E\,;p_1.\cdots.p_n, p \vdash \langle \sigma[r \mapsto V], e_1.\cdots.e_n, e\rangle$.*

The remaining lemmas are more specific to our study, so we outline their proofs. They say that values can be typed as not yielding, if they can be typed at all; that expressions that do not yield may be seen as yielding; and that `yield` can never appear in an evaluation context when the type system does not indicate yielding. They also provide an analysis of the possible forms of well-typed expressions.

Lemma 4. *If $E\,;p \vdash V : t$ then $E\,;\mathtt{NoYields} \vdash V : t$.*

This lemma holds simply because, in all the rules that can be used as the last one for typing a value ((Exp `unit`), (Exp `false`), (Exp `true`), (Exp x), and (Exp Fun)), the type system leaves the choice of effect completely unconstrained.

Lemma 5. *If $E\,;\mathtt{NoYields} \vdash e : t$ then $E\,;\mathtt{Yields} \vdash e : t$.*

The proof of Lemma 5 is by induction on the derivation of $E\,;\mathtt{NoYields} \vdash e : t$, with a case analysis on which rule is applied last. No rule forces a conclusion with `NoYields`: some rules where the conclusion may have effect `NoYields` (like (Exp Async) and (Exp Assert)) leave the choice of effect unconstrained, while others (like (Exp Appl) and (Exp Ref)) propagate the effect used in the hypotheses of the rule application. In the latter case, `Yields` can be used instead of `NoYields` also in the hypotheses of the rule application, by induction hypothesis and, in the case of (Exp Appl), because $q <: \mathtt{Yields}$ always holds.

Lemma 6. *It is never the case that* $E\,;\texttt{NoYields} \vdash \mathcal{P}[\ \texttt{yield}\ ] : t$.

The proof of Lemma 6 is by induction on typing derivations, with a case analysis on which rule is applied last.

- The cases of (Exp Unit), (Exp Bool `false`), (Exp Bool `true`), (Exp x), and (Exp Fun) are trivial, since the expressions typed there are values and cannot be the one in question.
- The case for (Exp Yield) is trivial because it gives an effect `Yields`.
- The cases of (Exp Appl), (Exp Ref), (Exp Deref), (Exp Set), and (Exp Block) are all by applications of the induction hypothesis, which are possible because the effects in the hypotheses of the rules are the same as the effects in their conclusions.
- The case for (Exp Async) is excluded because a context $\mathcal{P}$ cannot be of the form $\texttt{async}\ \mathcal{P}'$, so in this case $\mathcal{P}$ must be [ ], and $\texttt{async}\ \cdot$ cannot match `yield`.
- The case for (Exp Assert) is by application of the induction hypothesis, since the effects in the hypothesis of the rule is `NoYields`.

Lemma 7. *Suppose that* e *is a well-typed expression in which the only free variables are reference locations (with types of the form* $\texttt{Ref}\ t$*). Then* e *is a value or an expression of the form* $\mathcal{P}[\ f\ ]$*, where* f *has one of the forms* $(\lambda x.\, e')\ V$, $\texttt{ref}\ V$, $!r$, $r := V$, $\texttt{async}\ e'$, $\texttt{blockUntil true}$, $\texttt{blockUntil false}$, $\texttt{yield}$, *and* $\langle V \rangle$.

The proof of Lemma 7 is by induction on the typing of e, with a case analysis on the last rule in the typing derivation.

- In the cases of (Exp Unit), (Exp Bool `false`), (Exp Bool `true`), (Exp x), and (Exp Fun), e is a value.
- In the case of (Exp Appl), e cannot be a value. If $e_1\ e_2$ is well-typed, then e_1 and e_2 must be well-typed, and we apply the induction hypothesis to them. Suppose first that e_1 is a value. Because the type of e_1 must be a function type, e_1 must be of the form $\lambda x.\, e'$. (It cannot be a variable because reference locations do not have function types.) If e_2 is also a value V, we obtain that e is of the required form, with [ ] for $\mathcal{P}$. If e_2 is not a value, then it is of the form $\mathcal{P}'[\ f\ ]$, for an appropriate f, and we let $\mathcal{P}$ be $e_1\ \mathcal{P}'$. If e_1 is not a value, then it is of the form $\mathcal{P}'[\ f\ ]$, for an appropriate f, and we let $\mathcal{P}$ be $\mathcal{P}'\ e_2$.
- In the case of (Exp Ref), e cannot be a value. If $\texttt{ref}\ e_1$ is well-typed, then e_1 must be well-typed, and we apply the induction hypothesis to it. Suppose first that e_1 is a value. We obtain that e is of the required form, with [ ] for $\mathcal{P}$. If e_1 is not a value, then it is of the form $\mathcal{P}'[\ f\ ]$, for an appropriate f, and we let $\mathcal{P}$ be $\texttt{ref}\ \mathcal{P}'$.
- In the case of (Exp Deref), e cannot be a value. If $!e_1$ is well-typed, then e_1 must be well-typed, and we apply the induction hypothesis to it. Suppose first that e_1 is a value. Because the type of e_1 must be a reference type, e_1 must be a reference location r. We obtain that e is of the required form, with [ ] for $\mathcal{P}$. If e_1 is not a value, then it is of the form $\mathcal{P}'[\ f\ ]$, for an appropriate f, and we let $\mathcal{P}$ be $!\mathcal{P}'$.

– In the case of (Exp Set), e cannot be a value. If $e_1 := e_2$ is well-typed, then e_1 and e_2 must be well-typed, and we apply the induction hypothesis to them. Suppose first that e_1 is a value. Because the type of e_1 must be a reference type, e_1 must be a reference location r. If e_2 is also a value V, we obtain that e is of the required form, with $[\ ]$ for $\mathcal{P}$. If e_2 is not a value, then it is of the form $\mathcal{P}'[\ f\ ]$, for an appropriate f, and we let $\mathcal{P}$ be $r := \mathcal{P}'$. If e_1 is not a value, then it is of the form $\mathcal{P}'[\ f\ ]$, for an appropriate f, and we let $\mathcal{P}$ be $\mathcal{P}' := e_2$.
– The cases of (Exp Async) and (Exp Yield) are immediate, using the context $[\ ]$.
– In the case of (Exp Block), e cannot be a value. If `blockUntil` e_1 is well-typed, then e_1 must be well-typed, and we apply the induction hypothesis to it. Suppose first that e_1 is a value; according to the typing rules, it can be only `false` and `true`. (It cannot be a variable because reference locations do not have type `Bool`.) We obtain that e is of the required form, with $[\ ]$ for $\mathcal{P}$. If e_1 is not a value, then it is of the form $\mathcal{P}'[\ f\ ]$, for an appropriate f, and we let $\mathcal{P}$ be `blockUntil` $\mathcal{P}'$.
– In the case of (Exp Assert), e cannot be a value. If $\langle e_1 \rangle$ is well-typed, then e_1 must be well-typed, and we apply the induction hypothesis to it. Suppose first that e_1 is a value. We obtain that e is of the required form, with $[\ ]$ for $\mathcal{P}$. If e_1 is not a value, then it is of the form $\mathcal{P}'[\ f\ ]$, for an appropriate f, and we let $\mathcal{P}$ be $\langle \mathcal{P}' \rangle$.

Proof of Theorem 1. We prove that if $\langle \sigma, e_1.\cdots.e_n, e\rangle \longmapsto \langle \sigma', e'_1.\cdots.e'_{n'}, e'\rangle$ and $\langle \sigma, e_1.\cdots.e_n, e\rangle$ is well-typed then so is $\langle \sigma', e'_1.\cdots.e'_{n'}, e'\rangle$. The theorem follows immediately by induction.

The proof is by cases on the operational-semantics rule being applied. In each case, we show that if

$$E\,; p_1.\cdots.p_n, p \vdash \langle \sigma, e_1.\cdots.e_n, e\rangle$$

then

$$E'\,; p'_1.\cdots.p'_{n'}, p' \vdash \langle \sigma', e'_1.\cdots.e'_{n'}, e'\rangle$$

where, unless indicated otherwise, $E' = E$, $n' = n$, and $p'_i = p_i$ for $i = 1..n$. 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): The typing of $\langle \sigma, T, \mathcal{P}[\ (\lambda x.\, e)\ V\ ]\rangle$ 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 $E\,; p_0 \vdash \lambda x.\, e : t_1 \to^{q_0} t_0$ for some $q_0 <: p_0$ and $E\,; p_0 \vdash V : t_1$ for some t_1, and therefore $E, x : t_1\,; q_0 \vdash e : t_0$. By Lemma 5, $E, x : t_1\,; q_0 \vdash e : t_0$ and $q_0 <: p_0$ imply $E, x : t_1\,; p_0 \vdash e : t_0$. By Lemma 2, we obtain $E\,; p_0 \vdash e[V/x] : t_0$. By Lemma 1, we obtain a typing of $\langle \sigma, T, \mathcal{P}[\ e[V/x]\ ]\rangle$.
– (Trans Ref): The typing of $\langle \sigma, T, \mathcal{P}[\ \mathtt{ref}\ V\ ]\rangle$ must rely on (Exp Ref). Specifically, we must have $E\,; p_0 \vdash \mathtt{ref}\ V : \mathtt{Ref}\ t_0$ for some t_0 and p_0, and therefore

$E\,;p_0 \vdash V : t_0$. By Lemma 4, we obtain $E\,;\texttt{NoYields} \vdash V : t_0$. We extend E with $r : \texttt{Ref}\ t_0$. We can do this extension because $r \in \mathit{RefLoc} - \mathit{dom}(\sigma)$, hence $r \notin \mathit{dom}(E)$. By a weakening (adding $r : \texttt{Ref}\ t_0$ to E for typing $\langle \sigma, T, \mathcal{P}[\ \texttt{ref}\ V\ ]\rangle$) and Lemma 1, we obtain a typing of $\langle \sigma, T, \mathcal{P}[\ r\ ]\rangle$.

- (Trans Deref): The typing of $\langle \sigma, T, \mathcal{P}[\ !r\ ]\rangle$ 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 $E\,;p_0 \vdash r : \texttt{Ref}\ t_0$. Since r is a variable, its type must come from the environment E, so by hypothesis $E\,;\texttt{NoYields} \vdash V : t_0$ where $V = \sigma(r)$. By Lemma 5, we also have $E\,;\texttt{Yields} \vdash V : t_0$, which is useful in case p_0 is `Yields`. By Lemma 1, we obtain a typing for $\langle \sigma, T, \mathcal{P}[\ V\ ]\rangle$.
- (Trans Set): The typing of $\langle \sigma, T, \mathcal{P}[\ r := V\ ]\rangle$ must rely on (Exp Set). Specifically, we must have $E\,;p_0 \vdash r := V : \texttt{Unit}$ for some p_0, and therefore $E\,;p_0 \vdash V : t_0$ and $E\,;p_0 \vdash r : \texttt{Ref}\ t_0$ for some p_0. By Lemma 4, $E\,;p_0 \vdash V : t_0$ implies $E\,;\texttt{NoYields} \vdash V : t_0$. Since r is a variable, its type must come from the environment E. By Lemma 1, we can transform a typing of $\langle \sigma, T, \mathcal{P}[\ r := V\ ]\rangle$ into a typing of $\langle \sigma, T, \mathcal{P}[\ \texttt{unit}\ ]\rangle$, and since $E\,;\texttt{NoYields} \vdash V : t_0$ and $E(r) = \texttt{Ref}\ t_0$, we also obtain a typing of $\langle \sigma[r \mapsto V], T, \mathcal{P}[\ \texttt{unit}\ ]\rangle$ by Lemma 3.
- (Trans Async): The typing of $\langle \sigma, T, \mathcal{P}[\ \texttt{async}\ e\ ]\rangle$ must rely on (Exp Async). Specifically, we must have $E\,;p_0 \vdash \texttt{async}\ e : \texttt{Unit}$ for some p_0, and therefore that $E\,;q_0 \vdash e : \texttt{Unit}$ for some q_0. By Lemma 1, we can transform a typing of $\langle \sigma, T, \mathcal{P}[\ \texttt{async}\ e\ ]\rangle$ into a typing of $\mathcal{P}[\ \texttt{unit}\ ]$, and then into a typing of $\langle \sigma, T.e, \mathcal{P}[\ \texttt{unit}\ ]\rangle$ by letting $n' = n + 1$ and adding q_0 to the sequence of effects.
- (Trans Block): The typing of $\langle \sigma, T, \mathcal{P}[\ \texttt{blockUntil true}\ ]\rangle$ must rely on (Exp Block), specifically on a derivation of $E\,;p_0 \vdash \texttt{blockUntil true} : \texttt{Unit}$ for some p_0. By Lemma 1, we obtain a typing of $\langle \sigma, T, \mathcal{P}[\ \texttt{unit}\ ]\rangle$.
- (Trans Yield): This case requires a trivial rearrangement in the effects: $n' = n + 1$, $p'_{n+1} = p$, and $p' = \texttt{NoYields}$.
- (Trans Activate): This case requires a trivial rearrangement in the effects: $n' = n - 1$, and the effect p_i that corresponds to the expression e is skipped in $p'_1.\cdots.p'_{n'}$, and becomes p'.
- (Trans Assert): The typing of $\langle \sigma, T, \mathcal{P}[\ \langle V \rangle\ ]\rangle$ must rely on (Exp Assert). Specifically, we must have $E\,;p_0 \vdash \langle V \rangle : t_0$ for some t_0 and p_0, and $E\,;\texttt{NoYields} \vdash V : t_0$, so $E\,;p_0 \vdash V : t_0$ by Lemma 5. By Lemma 1, we obtain a typing of $\mathcal{P}[\ V\ ]$ and then of $\langle \sigma, T, \mathcal{P}[\ V\ ]\rangle$.

Proof of Theorem 2. By Theorem 1, if $\langle \sigma, T, e\rangle$ is well-typed then so are all the states reached in the computation $\langle \sigma, T, e\rangle \longmapsto^* \langle \sigma', T', e'\rangle$. Therefore, it suffices to prove that if $\langle \sigma, T, e\rangle$ is well-typed and $\langle \sigma, T, e\rangle \longmapsto \langle \sigma', T', e'\rangle$, then this transition is not an atomicity violation. The claim in the theorem then follows by induction.

So suppose that $\langle \sigma, T, e\rangle$ is well-typed and $\langle \sigma, T, e\rangle \longmapsto \langle \sigma', T', e'\rangle$. This transition could be an atomicity violation only if e is of the form $\mathcal{P}'[\ \langle \mathcal{P}''[\ \texttt{yield}\ ]\rangle\ ]$ for some $\mathcal{P}'$ and $\mathcal{P}''$. If $\langle \sigma, T, e\rangle$ is well-typed, then so is e, and therefore also $\langle \mathcal{P}''[\ \texttt{yield}\ ]\rangle$, because a state can be well-typed only if all its components and their subexpressions are well-typed. By the typing rule for assertions, the fact

that $\langle \mathcal{P}''[\ \texttt{yield}\ ] \rangle$ is well-typed implies that $E'\,;\texttt{NoYields} \vdash \mathcal{P}''[\ \texttt{yield}\ ] : t'$ for some E' and t'. We conclude by Lemma 6.

Proof of Theorem 3. According to Theorem 1, the state $\langle \sigma', T', e' \rangle$ is well-typed. Since the rules of the operational semantics do not introduce free variables other than reference locations, the only free variables in e' are reference locations. The desired conclusion follows from Lemma 8, given next.

Lemma 8. *If $\langle \sigma, T, e \rangle$ is well-typed, and the only free variables in e are reference locations, then:*

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

In order to prove Lemma 8, we apply Lemma 7 to e.

- If e is a value, then it must be `unit` because $\langle \sigma, T, e \rangle$ is well-typed and reference locations do not have type `Unit`. If T is empty, then we are in the third case. Otherwise, rule (Trans Activate) applies, and we are in the first case.
- If e is of the form $\mathcal{P}[\ \texttt{blockUntil false}\ ]$, then we are immediately in the second case.
- If e is of the form $\mathcal{P}[\ f\ ]$ where f is one of $(\lambda x.\, e')\ V$, $\texttt{ref}\ V$, $!r$, $r := V$, $\texttt{async}\ e'$, $\texttt{blockUntil true}$, $\texttt{yield}$, and $\langle V \rangle$, then (Trans Appl), (Trans Ref), (Trans Deref), (Trans Set), (Trans Async), (Trans Block), (Trans Yield), or (Trans Assert) apply, respectively, and we are in the first case again. In the case of (Trans Ref), we use that $\mathit{RefLoc} - \mathit{dom}(\sigma)$ is never empty. In the case of (Trans Deref), we rely on the condition that if $r \in \mathit{RefLoc}$ occurs free in e then $r \in \mathit{dom}(\sigma)$, and on the fact that σ maps reference locations to values.