A Model of Dynamic Separation for Transactional Memory

Martín Abadi [a,b] Tim Harris [a] Katherine F. Moore [a,c]

[a] *Microsoft Research*

[b] *University of California, Santa Cruz*

[c] *University of Washington*

Abstract

Dynamic separation is a new programming discipline for systems with transactional memory. We study it formally in the setting of a small calculus with transactions. We provide a precise formulation of dynamic separation and compare it with other programming disciplines. Furthermore, exploiting dynamic separation, we investigate some possible implementations of the calculus and we establish their correctness.

1 Introduction

Several designs and systems based on transactions aim to facilitate the writing of concurrent programs. In particular, software transactional memory (STM) appears as an intriguing alternative to locks and the related machinery for shared-memory concurrency [17]. STM implementations often allow transactions to execute in parallel, optimistically, detecting and resolving conflicts between transactions when they occur. Such implementations guarantee that transactions appear atomic with respect to other transactions, but not with respected to direct, non-transactional accesses to memory. This property has been termed "weak atomicity" [8], in contrast with the "strong atomicity" that programmers seem to expect, but which can be more challenging to provide.

Therefore, it is attractive to investigate programming disciplines under which the problematic discrepancy between "weak" implementations and "strong" semantics does not arise. In these disciplines, basically, transactional and non-transactional memory accesses should not be allowed to conflict. Much as in work on memory models (e.g., [5]), these disciplines can be seen as contracts between the language implementation and the programmer: if a program conforms to certain restrictions, then the language implementation must run it

with strong semantics. Such contracts should be "programmer-centric" [6]—formulated in terms of programs and their high-level semantics, not of implementation details. The selection of particular restrictions represents a tradeoff.

- Stronger restrictions give more flexibility to the implementation by requiring it to run fewer programs with strong semantics An example of such a restriction is the imposition of a static type system that strictly segregates transacted and non-transacted memory (e.g., [12,3,15]). This segregation often implies the need to copy data across these two parts of memory.
- Conversely, weaker restrictions give more flexibility to the programmer but may enable fewer implementation strategies. For example, violation-freedom prohibits only programs whose executions cause conflicts at run-time, according to a high-level, strong, small-step operational semantics [3] (see also [11,4,7]). Violation-freedom does not consider lower-level conflicts that may arise in implementations with optimistic concurrency; so these implementations may not run all violation-free programs with strong semantics, and may therefore be disallowed.

We are exploring a new programming discipline that we call dynamic separation. Its basic idea is to distinguish memory locations that should be accessed transactionally from those that should be accessed directly, allowing this distinction to evolve dynamically in the course of program execution. The programmer (perhaps with the assistance of tools) indicates transitions between these modes. Dynamic separation restricts only where data is actually accessed by a program, not how the data is reachable through references.

Dynamic separation is intermediate between violation-freedom and static separation. Like violation-freedom, it does not require copying between two memory regions; like static separation, on the other hand, it enables implementations with weak atomicity, optimistic concurrency, lazy conflict detection, and in-place updates. Indeed, dynamic separation is compatible with a range of transactional-memory implementations. Moreover, dynamic separation does not necessitate changes in how non-transactional code is compiled. This property makes transactions "pay-to-use" and lets non-transactional code rely on features not available for re-compilation (cf., e.g., [18]).

A companion paper [2] and a longer technical report [1] study dynamic separation informally. They provide a more detailed design rationale, an instantiation for C#, and some conceptually easy but useful refinements, in particular for read-only data. They also discuss implementations, describing our working implementation (done in the context of Bartok-STM [13]) and a variant that serves as a debugging tool for testing whether a program obeys the dynamic-separation discipline. As a case study, they examine the use of dynamic separation in the context of a concurrent web-proxy application built over an asynchronous IO library. Finally, they contain additional descriptions

of related informal work.

The present paper focuses on the formal definition and study of dynamic separation. It introduces constructs for dynamic separation, in the setting of a small calculus with transactions (Sections 2–4). It defines a dynamic-separation discipline and establishes precise comparisons with static separation and with violation-freedom (Section 5). Furthermore, it considers two possible lower-level implementations of the calculus (Sections 6 and 7). One of the implementations relies on two heaps, with marshaling between them. The other includes optimistic concurrency and some other challenging features; it models important aspects of our Bartok-STM implementation. We establish the correctness of both implementations: we prove that, if a program conforms to the dynamic-separation discipline, then the two implementations will run it with strong semantics. The Appendix contains proofs.

We present our results focusing on the Automatic Mutual Exclusion (AME) model [14,3] (Section 2). However, as we explain elsewhere [2,1], our approach applies also to other models for programming with transactions, for instance to TIC [19].

2 AME and the AME Calculus

In this section we describe the AME programming model and the AME calculus, a small language with AME constructs that serves as the setting of our formal study. This section is mostly an informal review; in addition it introduces the new constructs for indicating transitions between modes, named `protect` and `unprotect`, into the AME calculus. We postpone a formal semantics of the calculus to Section 4.

2.1 AME

AME distinguishes "protected" code, which executes within transactions, from ordinary "unprotected" code. Importantly, the default is protected code. Programmers are thus encouraged to use protected code as much as possible, leaving unprotected code primarily for interactions with legacy components. The intent is that this style of programming will lead to programs that are easier to understand and to maintain.

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 atomic fragments, discussed

below) in some serialized order. The invocation `async MethodName(<method arguments>)` creates an asynchronous call. The caller continues immediately after this invocation. In the conceptual serialization of the program, the asynchronous callee will be executed after the caller has completed. AME achieves concurrency by executing asynchronous calls in transactions, overlapping the execution of multiple calls, with roll-backs when conflicts occur. If a transaction initiates other asynchronous calls, their execution is deferred until the initiating transaction commits, and they are discarded if the initiating transaction aborts.

Methods may contain invocations of `yield()`, which break an asynchronous call into multiple atomic fragments, implemented by committing one transaction and starting a new one. With this addition, the overall execution of a program is guaranteed to be a serialization of its atomic fragments.

Methods may also contain statements of the form `blockUntil(<p>)`, where `p` is a predicate. From the programmer's perspective, an atomic fragment executes to completion only if all the predicates thus encountered in its execution evaluate to true. The implementation of `blockUntil(<p>)` does nothing if `p` holds; otherwise it aborts the current atomic fragment and retries it later.

In order to allow the use of legacy non-transactional code, AME provides block-structured `unprotected` sections. These must use existing mechanisms for synchronization. Before the execution of an `unprotected` section, the current atomic fragment is committed, and a new atomic fragment is started after the execution of the `unprotected` section.

AME allows memory locations to be accessed both transactionally and directly (that is, in `unprotected` sections). Dynamic separation amounts to requiring that, although each memory location may be accessed in both ways, the two kinds of accesses cannot occur simultaneously. In this paper we study the constructs `protect` and `unprotect` for supporting dynamic separation. Applying `protect` to a memory location indicates that it may be accessed only transactionally, and similarly applying `unprotect` to a memory location indicates that it may be accessed only directly.

2.2 The AME Calculus (with `protect` *and* `unprotect`*)*

The AME calculus is a small but expressive language that includes constructs for AME, higher-order functions, and imperative features. The following grammar defines the abstract syntax of the calculus, with the extensions required

for dynamic separation.

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

This 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, expressions include notations for function application ($e\ f$), 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). Expressions also include the new forms `protect` e and `unprotect` e, which evaluate e to a reference location, then make its value usable in transactions and outside transactions, respectively. We treat `yield` as syntactic sugar for `unprotected unit`. We write `let` $x = e$ `in` e' for $(\lambda x.\, e')\ e$, and write $e; e'$ for `let` $x = e$ `in` e' when x does not occur free in e'.

The grammar allows arbitrary nestings of `async`, `blockUntil`, `unprotected`, `protect`, `unprotect`, and the other constructs. Practical embodiments of AME need not be as liberal in these respects. In particular, `async` e may be limited to the case where e is a function call or method call, as in the informal review of Section 2.2. In addition, `protect` and `unprotect` may be required to occur only in unprotected code, in order to avoid the possibility of inconsistent updates to the protection modes, as explained further in Section 4. Although the grammar does not make this requirement, the semantics of Section 4 is designed so that `protect` and `unprotect` do not work in protected code.

We make a small technical restriction: in any expression of the form `async` e, any occurrences of `unprotected` are under a λ. Thus, with our syntactic sugar, we can write `async` (`unit`; `unprotected` e'), but not `async` (`unprotected` e'). More generally, we can write `async` (`unit`; e'), for any e'. This technical restriction roughly ensures that an unprotected computation is not the first thing that happens in an asynchronous computation. It is needed only for

Theorem 5.4, below.

3 An Example

This section presents an example, informally. Although this example is small and artificial, it serves to explain several aspects of our work. The example concerns the following code fragment:

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

This code first creates three reference locations, initialized to `false`, and binds x, y, and z to them, respectively. Then it forks two asynchronous executions. In one, it sets x to `true`. In the other, it sets x to `false`, checks that x holds `true`, then sets y to `true`. In addition, the code contains an unprotected section that checks that y holds `true`, then sets z to `true`.

In reasoning about such code, programmers (and tools) should be entitled to rely on the high-level semantics of the AME constructs, without considering their possible implementation details. According to this high-level semantics, the two asynchronous executions are serialized. Therefore, the predicate $!x$ in the second asynchronous execution can never hold, so $y :=$ `true` is unreachable. Hence the predicate $!y$ in the unprotected section can never hold either, so z will never be set to `true`. The formal semantics of Section 4 justifies this reasoning.

On the other hand, lower-level implementations, such as that modeled in Section 7, may exhibit different, surprising behavior. With optimistic concurrency, the two asynchronous executions may be attempted simultaneously. For efficiency, updates to reference locations may be done in place, not buffered. So, if the assignment $x :=$ `true` immediately follows the assignment $x :=$ `false`, then the predicate $!x$ in the second asynchronous execution will hold, and $y :=$ `true` will execute. After the assignment $x :=$ `true`, the execution of (`blockUntil` $(!x)$); $y :=$ `true` is a "zombie" [9], doomed to roll back. With lazy conflict detection, a conflict may not yet be apparent. With weak atomicity, moreover, the unprotected section has an opportunity to execute, and

the predicate $!y$ holds, so z will be set to `true`. When the two asynchronous executions attempt to commit, conflict detection will cause a roll-back of their effects on x and y, but not of the indirect effect on z. Therefore, the code may terminate with z holding `true`.

Despite the surprising behavior, we may want to allow such lower-level implementations because of their potential efficiency and compatibility with legacy code. So we may want to find criteria to exclude problematic programs. As indicated in the introduction, static separation is such a criterion; it statically segregates transacted and non-transacted memory. The code in our example does not obey static separation because (without dead-code elimination) y seems to be accessed both in a transaction and in the unprotected section. Unfortunately, static separation also forbids many reasonable code fragments, implying the need to marshal data back and forth between the two parts of memory.

Another possible criterion is violation-freedom. However, the code in our example is violation-free. In particular, according to the high-level semantics, there are no conflicting accesses to y at run-time, since $y :=$ `true` should never execute. Therefore, violation-freedom does not seem to be quite stringent enough to enable the use of some attractive implementation strategies.

Nevertheless, violation-free programs can often be instrumented with calls to `protect` and `unprotect` in order to conform to the dynamic-separation discipline. In this example, our particular formulation of dynamic separation requires adding two calls to `unprotect` in the last line of the code:

`unprotected` (`unprotect` y; `unprotect` z; (`blockUntil` $(!y)$); $z :=$ `true`)

Assuming that x, y, and z are initially in the mode where they are usable in transactions, we can reason that the placement of `unprotect` implies that x, y, and z are always used in the appropriate mode, so the code does conform to the dynamic-separation discipline. In this reasoning, we need to consider only the behavior of the code in the high-level semantics. Although the high-level semantics of `unprotect` is quite straightforward—and resembles that of no-op—an implementation of `unprotect` may do non-trivial work. Sections 6 and 7 provide two illustrations of this point, in the latter case modeling important aspects of our actual implementation in Bartok-STM. In particular, `unprotect` y may block while y is being written in a transaction, even if the transaction is a zombie. Moreover, updating y in a transaction may check that y is protected. Crucially, neither of these implementation refinements require any changes to non-transactional access to y. In combination, these refinements can prevent the problematic behavior of this code, guaranteeing that it runs correctly.

Zombies constitute only one of several problems in this area. Others include the so-called privatization and publication problems [10,9,21,3,20]. Although we do not discuss those in detail, our approach and our results address them as well. In particular, the correctness theorems below imply that publication and privatization idioms can execute correctly.

4 Semantics

The strong semantics of the AME calculus is a small-step operational semantics in which at most one transaction may take steps at any one time, and non-transactional code may take steps only when there is no current transaction taking steps [3]. We extend this strong semantics to the new constructs.

4.1 States

A state $\langle \sigma, \tau, T, e \rangle$ consists of a reference store σ, a protection state τ, a collection of expressions T (which we call "the pool"), and a distinguished active expression e. A reference store σ is a finite mapping of reference locations to values. Similarly, a protection state τ is a finite mapping of reference locations to protection modes, which we represent by the symbols `P` and `U`. It is a "history variable", in the sense that it is determined by the history of execution and does not influence this history. Reference locations are simply special kinds of variables that can be bound only by the respective store and protection state. We say that a reference location occurs in a partial function (such as a store or a protection state) if it is in the domain of the partial function. We write *RefLoc* for the set of reference locations; we assume that *RefLoc* is infinite. For every state $\langle \sigma, \tau, T, e \rangle$, we require that $dom(\sigma) = dom(\tau)$ and, if $r \in \mathit{RefLoc}$ occurs in $\langle \sigma, \tau, T, e \rangle$, then $r \in dom(\sigma)$. We set:

$$
\begin{aligned}
S \in \quad & \mathit{State} \subset \mathit{RefStore} \times \mathit{ProtState} \times \mathit{ExpSeq} \times \mathit{Exp} \\
\sigma \in \quad & \mathit{RefStore} = \mathit{RefLoc} \rightharpoonup \mathit{Value} \\
\tau \in \quad & \mathit{ProtState} = \mathit{RefLoc} \rightharpoonup \{\mathtt{P}, \mathtt{U}\} \\
r \in \quad & \mathit{RefLoc} \subset \mathit{Var} \\
T \in \quad & \mathit{ExpSeq} = \mathit{Exp}^*
\end{aligned}
$$

4.2 Steps

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

$$\begin{aligned}
\mathcal{P} &= [\ ] \mid \mathcal{P}\ e \mid V\ \mathcal{P} \mid \texttt{ref}\ \mathcal{P} \mid !\mathcal{P} \mid \mathcal{P} := e \mid r := \mathcal{P} \\
&\quad \mid \texttt{blockUntil}\ \mathcal{P} \mid \texttt{protect}\ \mathcal{P} \mid \texttt{unprotect}\ \mathcal{P} \\
\mathcal{U} &= \texttt{unprotected}\ \mathcal{E} \mid \mathcal{U}\ e \mid V\ \mathcal{U} \mid \texttt{ref}\ \mathcal{U} \mid !\mathcal{U} \mid \mathcal{U} := e \mid r := \mathcal{U} \\
&\quad \mid \texttt{blockUntil}\ \mathcal{U} \mid \texttt{protect}\ \mathcal{U} \mid \texttt{unprotect}\ \mathcal{U} \\
\mathcal{E} &= [\ ] \mid \mathcal{E}\ e \mid V\ \mathcal{E} \mid \texttt{ref}\ \mathcal{E} \mid !\mathcal{E} \mid \mathcal{E} := e \mid r := \mathcal{E} \\
&\quad \mid \texttt{blockUntil}\ \mathcal{E} \mid \texttt{unprotected}\ \mathcal{E} \mid \texttt{protect}\ \mathcal{E} \mid \texttt{unprotect}\ \mathcal{E}
\end{aligned}$$

A context $\mathcal{E}$ is a general evaluation context; a context $\mathcal{U}$ is one where the hole is under `unprotected`; a context $\mathcal{P}$ is one where it is not. Note that neither $\lambda x.\ [\ ]$ nor `async` [ ] are evaluation contexts: both abstraction and forking delay evaluation.

Figure 1 gives rules that specify the transition relation that takes execution from one state to the next. In these rules, we write $e[V/x]$ for the result of the capture-free substitution of V for x in e, and write $\sigma[r \mapsto V]$ for the store that agrees with σ except at r, which is mapped to V. The subscript s in $\longmapsto_s$ indicates that this is a strong semantics. The rules whose names have a "P" suffix (like (Trans Ref P)$_s$) correspond to "protected" execution of the active expression in a $\mathcal{P}$ context. The rules whose names have a "U" suffix (like (Trans Ref U)$_s$) correspond to "unprotected" execution in a $\mathcal{U}$ context in the pool; in all these, the active expression must be `unit`. Some rules (such as (Trans Activate)$_s$) apply only in the "protected world" or only in the "unprotected world", and do not have a counterpart in the other world, so we do not include a "P" or a "U" suffix in their names.

In rules (Trans Ref P)$_s$ and (Trans Ref U)$_s$, the reference-allocation construct `ref` e initializes the new location's mode to `P` (when allocating inside a transaction) or to `U` (otherwise). In rules (Trans DynPr)$_s$ and (Trans DynUn)$_s$, the new constructs `protect` and `unprotect` set the mode to `P` and to `U` respectively. It is not an error to call `protect` on a reference location already in mode `P`. Similarly, it is not an error to call `unprotect` on a reference location already in mode `U`. This design choice enables a broader range of implementations, as discussed in our companion paper.

According to the rules, `protect` and `unprotect` work only outside transac-

$$\langle \sigma, \tau, T, \mathcal{P}[\ (\lambda x.\ e)\ V\ ] \rangle \longmapsto_s \langle \sigma, \tau, T, \mathcal{P}[\ e[V/x]\ ] \rangle \quad \text{(Trans Appl P)}_s$$

$$\langle \sigma, \tau, T.\mathcal{U}[\ (\lambda x.\ e)\ V\ ].T', \texttt{unit} \rangle \longmapsto_s \langle \sigma, \tau, T.\mathcal{U}[\ e[V/x]\ ].T', \texttt{unit} \rangle \quad \text{(Trans Appl U)}_s$$

$$\langle \sigma, \tau, T, \mathcal{P}[\ \texttt{ref}\ V\ ] \rangle \longmapsto_s \langle \sigma[r \mapsto V], \tau[r \mapsto \texttt{P}], T, \mathcal{P}[\ r\ ] \rangle \quad \text{(Trans Ref P)}_s$$
$$\text{if } r \in \mathit{RefLoc} - \mathit{dom}(\sigma)$$

$$\langle \sigma, \tau, T.\mathcal{U}[\ \texttt{ref}\ V\ ].T', \texttt{unit} \rangle \longmapsto_s \langle \sigma[r \mapsto V], \tau[r \mapsto \texttt{U}], T.\mathcal{U}[\ r\ ].T', \texttt{unit} \rangle \quad \text{(Trans Ref U)}_s$$
$$\text{if } r \in \mathit{RefLoc} - \mathit{dom}(\sigma)$$

$$\langle \sigma, \tau, T, \mathcal{P}[\ !r\ ] \rangle \longmapsto_s \langle \sigma, \tau, T, \mathcal{P}[\ V\ ] \rangle \quad \text{(Trans Deref P)}_s$$
$$\text{if } \sigma(r) = V$$

$$\langle \sigma, \tau, T.\mathcal{U}[\ !r\ ].T', \texttt{unit} \rangle \longmapsto_s \langle \sigma, \tau, T.\mathcal{U}[\ V\ ].T', \texttt{unit} \rangle \quad \text{(Trans Deref U)}_s$$
$$\text{if } \sigma(r) = V$$

$$\langle \sigma, \tau, T, \mathcal{P}[\ r := V\ ] \rangle \longmapsto_s \langle \sigma[r \mapsto V], \tau, T, \mathcal{P}[\ \texttt{unit}\ ] \rangle \quad \text{(Trans Set P)}_s$$

$$\langle \sigma, \tau, T.\mathcal{U}[\ r := V\ ].T', \texttt{unit} \rangle \longmapsto_s \langle \sigma[r \mapsto V], \tau, T.\mathcal{U}[\ \texttt{unit}\ ].T', \texttt{unit} \rangle \quad \text{(Trans Set U)}_s$$

$$\langle \sigma, \tau, T, \mathcal{P}[\ \texttt{async}\ e\ ] \rangle \longmapsto_s \langle \sigma, \tau, e.T, \mathcal{P}[\ \texttt{unit}\ ] \rangle \quad \text{(Trans Async P)}_s$$

$$\langle \sigma, \tau, T.\mathcal{U}[\ \texttt{async}\ e\ ].T', \texttt{unit} \rangle \longmapsto_s \langle \sigma, \tau, e.T.\mathcal{U}[\ \texttt{unit}\ ].T', \texttt{unit} \rangle \quad \text{(Trans Async U)}_s$$

$$\langle \sigma, \tau, T, \mathcal{P}[\ \texttt{blockUntil true}\ ] \rangle \longmapsto_s \langle \sigma, \tau, T, \mathcal{P}[\ \texttt{unit}\ ] \rangle \quad \text{(Trans Block P)}_s$$

$$\langle \sigma, \tau, T.\mathcal{U}[\ \texttt{blockUntil true}\ ].T', \texttt{unit} \rangle \longmapsto_s \langle \sigma, \tau, T.\mathcal{U}[\ \texttt{unit}\ ].T', \texttt{unit} \rangle \quad \text{(Trans Block U)}_s$$

$$\langle \sigma, \tau, T, \mathcal{P}[\ \texttt{unprotected}\ e\ ] \rangle \longmapsto_s \langle \sigma, \tau, T.\mathcal{P}[\ \texttt{unprotected}\ e\ ], \texttt{unit} \rangle \quad \text{(Trans Unprotect)}_s$$

$$\langle \sigma, \tau, T.\mathcal{E}[\ \texttt{unprotected}\ V\ ].T', \texttt{unit} \rangle \longmapsto_s \langle \sigma, \tau, T.\mathcal{E}[\ V\ ].T', \texttt{unit} \rangle \quad \text{(Trans Close)}_s$$

$$\langle \sigma, \tau, T.e.T', \texttt{unit} \rangle \longmapsto_s \langle \sigma, \tau, T.T', e \rangle \quad \text{(Trans Activate)}_s$$

$$\langle \sigma, \tau, T.\mathcal{U}[\ \texttt{protect}\ r\ ].T', \texttt{unit} \rangle \longmapsto_s \langle \sigma, \tau[r \mapsto \texttt{P}], T.\mathcal{U}[\ r\ ].T', \texttt{unit} \rangle \quad \text{(Trans DynPr)}_s$$

$$\langle \sigma, \tau, T.\mathcal{U}[\ \texttt{unprotect}\ r\ ].T', \texttt{unit} \rangle \longmapsto_s \langle \sigma, \tau[r \mapsto \texttt{U}], T.\mathcal{U}[\ r\ ].T', \texttt{unit} \rangle \quad \text{(Trans DynUn)}_s$$

Fig. 1. Transition rules with dynamic separation.

tions. They get stuck otherwise. Fundamentally, we do not want to rely on `protect` and `unprotect` in transactions because of questionable interactions, such as the possibility of zombie updates to the protection state.

5 The Dynamic-Separation Discipline

We give a precise definition of dynamic separation. We also establish results that relate dynamic separation to static separation and to violation-freedom.

5.1 Definition

The definition of dynamic separation says that, in the course of an execution, reads and writes to a reference location should happen only if the protection state of the reference location is consistent with the context of the operation. The definition is intended to constrain expressions, but more generally it applies to initial states of executions.

Given a state $\langle \sigma, \tau, T, e \rangle$, a read or a write may occur in two cases:

- e is of the form $\mathcal{P}[\ !r\ ]$ or $\mathcal{P}[\ r := V\ ]$; or
- $e = \texttt{unit}$ and T contains an expression of the form $\mathcal{U}[\ !r\ ]$ or $\mathcal{U}[\ r := V\ ]$.

Accordingly, we say that a state $\langle \sigma, \tau, T, e \rangle$ is locally good when:

- if e is of the form $\mathcal{P}[\ !r\ ]$ or $\mathcal{P}[\ r := V\ ]$, then $\tau(r) = \texttt{P}$;
- if $e = \texttt{unit}$ and T contains an expression of the form $\mathcal{U}[\ !r\ ]$ or $\mathcal{U}[\ r := V\ ]$, then $\tau(r) = \texttt{U}$.

Further, we say that a state S obeys the dynamic-separation discipline, and write $\mathcal{DS}(S)$, if whenever $S \longmapsto_s^* S'$, the state S' is locally good.

In sum, a state S obeys the dynamic-separation discipline if, in S, reads or writes to a reference location r can happen only if r's protection state (`P` or `U`) is consistent with the context (transactional or not, respectively) of the operation, and if the same is true for any state reachable from S.

5.2 A Variant

A stronger notion of dynamic separation is also worth considering, because it is closer to violation-freedom (as discussed below). This stronger notion is defined much like dynamic separation. First, we say that a state $\langle \sigma, \tau, T, e \rangle$ is locally good' when:

- if e is of the form $\mathcal{P}[\ !r\ ]$ or $\mathcal{P}[\ r := V\ ]$, then $\tau(r) = \texttt{P}$;
- if T contains an expression of the form $\mathcal{U}[\ !r\ ]$ or $\mathcal{U}[\ r := V\ ]$ then $\tau(r) = \texttt{U}$.

We say that a state S obeys the dynamic-separation' discipline, and write $\mathcal{DS}'(S)$, if whenever $S \longmapsto_s^* S'$, the state S' is locally good'. Obviously, $\mathcal{DS}'(S)$ always implies $\mathcal{DS}(S)$, but the converse implication need not hold. Crucially, the definition of locally good' does not assume $e = \texttt{unit}$ in its second clause; when $e \neq \texttt{unit}$, the memory access indicated in T cannot actually proceed according to the strong semantics.

Although dynamic separation does not imply dynamic separation', this implication holds when the active expression is `unit`:

Lemma 5.1 *If* $\mathcal{DS}(\langle \sigma, \tau, T, \texttt{unit} \rangle)$ *then* $\mathcal{DS}'(\langle \sigma, \tau, T, \texttt{unit} \rangle)$.

In other words, $\mathcal{DS}(S)$ and $\mathcal{DS}'(S)$ are equivalent when S is of the form $\langle \sigma, \tau, T, \texttt{unit} \rangle$. Despite its apparent simplicity, this lemma is not an immediate consequence of the definitions of locally good and locally good'. The lemma would be trivial if computation preserved that the active expression is `unit`, but it does not.

5.3 Comparison with Static Separation

Static separation can be defined as a type system; its details are straightforward, and for AME they are given in [3, Section 6.2]. There, the judgment $E \vdash \langle \sigma, T, e \rangle$ says that the state $\langle \sigma, T, e \rangle$ obeys the static-separation discipline in a typing environment E, which gives types of the form $\texttt{Ref}_\texttt{P}\ t$ or $\texttt{Ref}_\texttt{U}\ t$ for the free reference locations of the state. The state does not include a protection state τ, since separation is static. Given E, however, we write τ_E for the protection state that maps each reference location to `P` or `U` according to its type in E.

Our first theorem about static separation says that the transition relation of the strong semantics ($\longmapsto_s$) preserves typability. The theorem also characterizes the corresponding typing environments. It can be seen as an extension of Theorem 6.1 of [3], which concerns preservation of typability for a language without `protect` and `unprotect`, and for an operational semantics without protection states.

Theorem 5.2 *If* $E \vdash \langle \sigma, T, e \rangle$, *and*

$$\langle \sigma, \tau_E, T, e \rangle \longmapsto_s^* \langle \sigma', \tau', T', e' \rangle$$

then there is some E' *such that:*

- $E' \vdash \langle \sigma', T', e' \rangle$, *and*
- $\tau' = \tau_{E'}$

We obtain that static separation implies dynamic separation:

Theorem 5.3 *If* $E \vdash \langle \sigma, T, e \rangle$ *then* $\mathcal{DS}(\langle \sigma, \tau_E, T, e \rangle)$.

The converse of this theorem is false, not only because of possible occurrences of `protect` and `unprotect` but also because of examples like that of Section 3.

5.4 Comparison with Violation-Freedom

As discussed above, violation-freedom is a condition that prohibits programs whose executions cause certain conflicts at run-time. More precisely, we say that a state $\langle \sigma, \tau, T, e \rangle$ has a violation on r when:

- e is of the form $\mathcal{P}[\ e'\ ]$,
- T contains an expression of the form $\mathcal{U}[\ e''\ ]$,
- e' and e'' are of the form $!r$ or $r := V$ for some V, and at least one is of the latter form.

(Note that the second of these clauses does not require $e = \texttt{unit}$, unlike the corresponding part of the definition of local goodness, but like the corresponding part of the definition of local goodness'.) We say that a state S obeys the violation-freedom discipline, and write $\mathcal{VF}(S)$, if whenever $S \longmapsto_s^* S'$, the state S' does not have violations on any r.

In general, dynamic separation is not sufficient for violation-freedom. For instance, the state

$$\langle \emptyset[r \mapsto \texttt{false}], \emptyset[r \mapsto \texttt{P}], \texttt{unprotected}\ (r := \texttt{true}), \texttt{blockUntil}\ !r \rangle$$

obeys the dynamic-separation discipline, but has an obvious violation on r (and it is not locally good'). This violation never leads to an actual concurrent access under the strong semantics.

Dynamic separation does however imply violation-freedom for initial states of the form $\langle \sigma, \tau, T, \texttt{unit} \rangle$, in which there is no active transaction—but of course a transaction may be activated. We regard this result, together with Theorem 5.3, as proof of our informal statement that dynamic separation is intermediate between violation-freedom and static separation.

Theorem 5.4 *If* $\mathcal{DS}(\langle \sigma, \tau, T, \texttt{unit} \rangle)$ *then* $\mathcal{VF}(\langle \sigma, \tau, T, \texttt{unit} \rangle)$.

This theorem is an immediate consequence of Lemma 5.1 and of the following Lemma, which applies even when the active expression is not `unit`:

Lemma 5.5 *If* $\mathcal{DS}'(\langle \sigma, \tau, T, e \rangle)$ *then* $\mathcal{VF}(\langle \sigma, \tau, T, e \rangle)$.

Conversely, violation-freedom is not a sufficient condition for dynamic separation, for at least two reasons:

- Most obviously, violation-freedom does not require the use of explicit calls to `protect` and `unprotect`.
- In addition, violation-freedom does not constrain read-read concurrency, while dynamic separation does.

One may try to address the first reason by adding calls to `protect` and `unprotect`, and the second by strengthening violation-freedom so that it also constrains read-read concurrency. With this strengthening, we have explored a method for taking a violation-free expression and adding calls to `protect` and `unprotect` so as to make it obey dynamic separation. Basically, we bracket each non-transactional memory access between a call to `unprotect` and a call to `protect`. We omit the details of our method, but briefly note its two main assumptions:

- The method requires the absence of race conditions in unprotected computations, because race conditions could cause instrumentation (the calls to `protect` and `unprotect`) to work incorrectly.
- It also assumes that we can distinguish transactional and non-transactional code at instrumentation time. Code duplication can make this task trivial.

6 An Implementation with Two Heaps

In this section, we consider an abstract machine with two separate heaps accessed by transactional and non-transactional code, respectively. The constructs `protect` and `unprotect` marshal between these heaps. Although this two-heap scheme is not particularly efficient, it is reminiscent of some practical systems that use different data formats in transactional and non-transactional code. It is also an interesting approximation of a static-separation regime, and illustrates that `protect` and `unprotect` may do more than in the high-level semantics of Figure 1. Still, for expressions that obey the dynamic-separation discipline, we prove that this two-heap implementation respects the high-level semantics.

6.1 Operational Semantics

We define the two-heap implementation as a lower-level semantics, in the style of that of Section 4 though with some additional intricacies.

$$\langle \sigma_1, \sigma_2, \tau, T, \mathcal{P}[\ (\lambda x.\ e)\ V\ ] \rangle \longmapsto_t \langle \sigma_1, \sigma_2, \tau, T, \mathcal{P}[\ e[V/x]\ ] \rangle \quad \text{(Trans Appl P)}_t$$

$$\langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ (\lambda x.\ e)\ V\ ].T', \texttt{unit} \rangle \longmapsto_t \langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ e[V/x]\ ].T', \texttt{unit} \rangle \quad \text{(Trans Appl U)}_t$$

$$\langle \sigma_1, \sigma_2, \tau, T, \mathcal{P}[\ \texttt{ref}\ V\ ] \rangle \longmapsto_t \langle \sigma_1[r \mapsto V], \sigma_2[r \mapsto V], \tau[r \mapsto \texttt{P}], T, \mathcal{P}[\ r\ ] \rangle \quad \text{(Trans Ref P)}_t$$
if $r \in \mathit{RefLoc} - \mathit{dom}(\sigma_1)$

$$\langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ \texttt{ref}\ V\ ].T', \texttt{unit} \rangle \longmapsto_t \langle \sigma_1[r \mapsto V], \sigma_2[r \mapsto V], \tau[r \mapsto \texttt{U}], T.\mathcal{U}[\ r\ ].T', \texttt{unit} \rangle \quad \text{(Trans Ref U)}_t$$
if $r \in \mathit{RefLoc} - \mathit{dom}(\sigma_1)$

$$\langle \sigma_1, \sigma_2, \tau, T, \mathcal{P}[\ !r\ ] \rangle \longmapsto_t \langle \sigma_1, \sigma_2, \tau, T, \mathcal{P}[\ V\ ] \rangle \quad \text{(Trans Deref P)}_t$$
if $\sigma_1(r) = V$

$$\langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ !r\ ].T', \texttt{unit} \rangle \longmapsto_t \langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ V\ ].T', \texttt{unit} \rangle \quad \text{(Trans Deref U)}_t$$
if $\sigma_2(r) = V$

$$\langle \sigma_1, \sigma_2, \tau, T, \mathcal{P}[\ r := V\ ] \rangle \longmapsto_t \langle \sigma_1[r \mapsto V], \sigma_2, \tau, T, \mathcal{P}[\ \texttt{unit}\ ] \rangle \quad \text{(Trans Set P)}_t$$

$$\langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ r := V\ ].T', \texttt{unit} \rangle \longmapsto_t \langle \sigma_1, \sigma_2[r \mapsto V], \tau, T.\mathcal{U}[\ \texttt{unit}\ ].T', \texttt{unit} \rangle \quad \text{(Trans Set U)}_t$$

$$\langle \sigma_1, \sigma_2, \tau, T, \mathcal{P}[\ \texttt{async}\ e\ ] \rangle \longmapsto_t \langle \sigma_1, \sigma_2, \tau, e.T, \mathcal{P}[\ \texttt{unit}\ ] \rangle \quad \text{(Trans Async P)}_t$$

$$\langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ \texttt{async}\ e\ ].T', \texttt{unit} \rangle \longmapsto_t \langle \sigma_1, \sigma_2, \tau, e.T.\mathcal{U}[\ \texttt{unit}\ ].T', \texttt{unit} \rangle \quad \text{(Trans Async U)}_t$$

$$\langle \sigma_1, \sigma_2, \tau, T, \mathcal{P}[\ \texttt{blockUntil true}\ ] \rangle \longmapsto_t \langle \sigma_1, \sigma_2, \tau, T, \mathcal{P}[\ \texttt{unit}\ ] \rangle \quad \text{(Trans Block P)}_t$$

$$\langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ \texttt{blockUntil true}\ ].T', \texttt{unit} \rangle \longmapsto_t \langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ \texttt{unit}\ ].T', \texttt{unit} \rangle \quad \text{(Trans Block U)}_t$$

$$\langle \sigma_1, \sigma_2, \tau, T, \mathcal{P}[\ \texttt{unprotected}\ e\ ] \rangle \longmapsto_t \langle \sigma_1, \sigma_2, \tau, T.\mathcal{P}[\ \texttt{unprotected}\ e\ ], \texttt{unit} \rangle \quad \text{(Trans Unprotect)}_t$$

$$\langle \sigma_1, \sigma_2, \tau, T.\mathcal{E}[\ \texttt{unprotected}\ V\ ].T', \texttt{unit} \rangle \longmapsto_t \langle \sigma_1, \sigma_2, \tau, T.\mathcal{E}[\ V\ ].T', \texttt{unit} \rangle \quad \text{(Trans Close)}_t$$

$$\langle \sigma_1, \sigma_2, \tau, T.e.T', \texttt{unit} \rangle \longmapsto_t \langle \sigma_1, \sigma_2, \tau, T.T', e \rangle \quad \text{(Trans Activate)}_t$$

$$\langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ \texttt{protect}\ r\ ].T', \texttt{unit} \rangle \longmapsto_t \langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ r\ ].T', \texttt{unit} \rangle \quad \text{(Trans DynPr (1))}_t$$
if $\tau(r) = \texttt{P}$

$$\langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ \texttt{protect}\ r\ ].T', \texttt{unit} \rangle \longmapsto_t \langle \sigma_1[r \mapsto \sigma_2(r)], \sigma_2, \tau[r \mapsto \texttt{P}], T.\mathcal{U}[\ r\ ].T', \texttt{unit} \rangle \quad \text{(Trans DynPr (2))}_t$$
if $\tau(r) = \texttt{U}$

$$\langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ \texttt{unprotect}\ r\ ].T', \texttt{unit} \rangle \longmapsto_t \langle \sigma_1, \sigma_2[r \mapsto \sigma_1(r)], \tau[r \mapsto \texttt{U}], T.\mathcal{U}[\ r\ ].T', \texttt{unit} \rangle \quad \text{(Trans DynUn (1))}_t$$
if $\tau(r) = \texttt{P}$

$$\langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ \texttt{unprotect}\ r\ ].T', \texttt{unit} \rangle \longmapsto_t \langle \sigma_1, \sigma_2, \tau, T.\mathcal{U}[\ r\ ].T', \texttt{unit} \rangle \quad \text{(Trans DynUn (2))}_t$$
if $\tau(r) = \texttt{U}$

Fig. 2. Transition rules with two heaps.

6.1.1 States.

The components of a state are much like those in Section 4, except that there are two reference stores rather than one. A state $\langle \sigma_1, \sigma_2, \tau, T, e \rangle$ consists of two reference stores σ_1 and σ_2, a protection state τ, a collection of expressions T, and a distinguished active expression e. Intuitively, σ_1 and σ_2 are for the use of transactional and non-transactional code, respectively. We require that $dom(\sigma_1) = dom(\sigma_2) = dom(\tau)$ and that, if $r \in \mathit{RefLoc}$ occurs in the state, then $r \in dom(\sigma_1)$. So we set:

$$S \in \mathit{State} \subset \mathit{RefStore} \times \mathit{RefStore} \times \mathit{ProtState} \times \mathit{ExpSeq} \times \mathit{Exp}$$

6.1.2 Steps.

Figure 2 gives rules that specify the transition relation of this semantics. According to these rules, `ref` e sets the protection state of a new reference location r and initializes the contents of r in each of the reference stores. Initializing the contents in the appropriate reference store would suffice, provided r is added to the domain of both reference stores. While reading or writing a location, the context in which an expression executes determines which reference store it accesses. Finally, `protect` r and `unprotect` r perform marshaling, as follows. If r already has the desired protection state, then no copying is required. (In fact, copying could overwrite fresh contents with stale ones.) Otherwise, r's contents are copied from one reference store to the other.

6.2 Correctness

The two-heap implementation is correct under the dynamic-separation discipline, in the following sense:

Theorem 6.1 *Assume that* $\mathcal{DS}(\langle \sigma, \tau, T, e \rangle)$, *that* $dom(\sigma) = dom(\sigma_1) = dom(\sigma_2)$, *and that* $\sigma_1(r) = \sigma(r)$ *if* $\tau(r) = \mathtt{P}$ *and* $\sigma_2(r) = \sigma(r)$ *if* $\tau(r) = \mathtt{U}$. *Consider a computation with two heaps:*

$$\langle \sigma_1, \sigma_2, \tau, T, e \rangle \longmapsto_t^* \langle \sigma_1', \sigma_2', \tau', T', e' \rangle$$

Then there is a computation:

$$\langle \sigma, \tau, T, e \rangle \longmapsto_s^* \langle \sigma', \tau', T', e' \rangle$$

for some σ' *such that* $dom(\sigma') = dom(\sigma_1') = dom(\sigma_2')$ *and, for every* $r \in dom(\sigma')$, *if* $\tau'(r) = \mathtt{P}$, *then* $\sigma_1'(r) = \sigma'(r)$, *and if* $\tau'(r) = \mathtt{U}$, *then* $\sigma_2'(r) = \sigma'(r)$.

This simulation result implies that the contents of a reference location r is always correct in the reference store that corresponds to r's current protection state. The dynamic-separation hypothesis is essential: it is required for extending the simulation in the cases of (Trans Deref $\dots$)$_t$ and (Trans Set $\dots$)$_t$. Without it, the execution with two heaps may produce incorrect results.

7 An Implementation with Optimistic Concurrency

Going further, we treat a lower-level implementation in which multiple transactions execute simultaneously, with roll-backs in case of conflict. This implementation is based on one studied in our previous work [3], with the addition of dynamic separation. As explained there, various refinements are possible, but they are not necessary for our present purposes. Our goal is to show how dynamic separation works (correctly) in a setting with realistic, challenging features such as in-place updates (e.g., [16]). The model developed in this section is an abstract version of our actual implementation in Bartok-STM.

7.1 Operational Semantics

Again, we define the implementation as a lower-level semantics.

7.1.1 States.

States become more complex for this semantics. In addition to the components σ, τ, and T that appear in the earlier semantics, we add constructs for roll-back and optimistic concurrency. In order to support roll-back, we maintain a log l of the reference locations that have been modified, with their corresponding original values. In the case of roll-back, we use the log to restore these values in the reference store. For optimistic concurrency, we have a list of tuples instead of a single active expression. Each of the tuples is called a try, and consists of the following components:

- an active expression e,
- another expression f from which e was obtained (its "origin"),
- 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.

For every state $\langle \sigma, \tau, T, O, l \rangle$, we require that $dom(\sigma) = dom(\tau)$ and that, if $r \in \mathit{RefLoc}$ occurs in the state, then $r \in dom(\sigma)$. We set:

$$
\begin{aligned}
S \in \quad & \mathit{State} \subset \mathit{RefStore} \times \mathit{ProtState} \times \mathit{ExpSeq} \times \mathit{TrySeq} \times \mathit{Log} \\
\sigma \in \quad & \mathit{RefStore} = \mathit{RefLoc} \rightharpoonup \mathit{Value} \\
\tau \in \quad & \mathit{ProtState} = \mathit{RefLoc} \rightharpoonup \{\mathtt{P}, \mathtt{U}\} \\
l \in \quad & \mathit{Log} = \mathit{RefLoc} \rightharpoonup \mathit{Value} \\
r \in \quad & \mathit{RefLoc} \subset \mathit{Var} \\
T, P \in \quad & \mathit{ExpSeq} = \mathit{Exp}^* \\
O \in \quad & \mathit{TrySeq} = \mathit{Try}^* \\
d \in \quad & \mathit{Try} = \mathit{Exp} \times \mathit{Exp} \times \mathit{Accesses} \times \mathit{ExpSeq} \\
a \in \quad & \mathit{Accesses} = \mathit{RefLoc}^*
\end{aligned}
$$

7.1.2 Steps.

Figure 3 gives the rules of this semantics, relying on these 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.
- 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 store that agrees with l on $dom(l)$, and with σ elsewhere.

These rules allow for conflicts to be detected as soon as they occur, but they do not require it. In other words, conflict detection may be eager or lazy: the rules do not impose a particular strategy in this respect. The rules simply allow undo to happen at any point (whether or not there is a conflict). For simplicity, undos are atomic and affect all current transactions. Moreover, conflict detection does not distinguish reads and writes, and the log used for undos does not contain timestamps or other information that might support selective undos. In contrast, actual STM implementations typically resolve conflicts by aborting some transactions and committing others. Thus, many refinements of our rules are possible. These aspects of the semantics are explained at more length in our previous work. Here we focus on the new ones, namely those related to dynamic separation.

Rule (Trans DynUn)$_o$ requires that, when a reference location is unprotected, it is not being written by any try. This restriction is a formalization of one

present in our Bartok-STM implementation (where "being written" means, more specifically, "open for update"). The restriction on (Trans DynUn)$_o$ can be satisfied by performing an undo. However, an undo is never forced to happen. Indeed, as explained above, the rules allow undo to happen at any point—possibly but not necessarily when there is a conflict.

There is no corresponding subtlety in rule (Trans DynPr)$_o$. Bartok-STM employs a more elaborate version of this rule in order to allow compiler optimizations that reorder accesses.

When writing to a reference location from within a transaction (rule (Trans Set P)$_o$), the protection state of that reference location is verified. Even with dynamic separation, this check is essential for correctness because of the possibility of zombie transactions (which could otherwise modify all of memory arbitrarily). On the other hand, a check is not needed for reads (rule (Trans Deref P)$_o$), nor for accesses in unprotected code (rules (Trans Deref U)$_o$ and (Trans Set U)$_o$). These features of the rules correspond to important aspects of our Bartok-STM implementation, which aims to allow the re-use of legacy code without instrumentation.

7.2 Correctness

The implementation with optimistic concurrency is correct with respect to the strong semantics of Section 4, in the following sense:

Theorem 7.1 *Assume that* $\mathcal{DS}(\langle \sigma, \tau, T, \texttt{unit} \rangle)$. *Consider a computation:*

$$\langle \sigma, \tau, T, \emptyset, \emptyset \rangle \longmapsto_o^* \langle \sigma', \tau', T', \emptyset, \emptyset \rangle$$

Then there is a computation:

$$\langle \sigma, \tau, T, \texttt{unit} \rangle \longmapsto_s^* \langle \sigma'', \tau'', T'', \texttt{unit} \rangle$$

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

Much as for Theorem 6.1, the dynamic-separation assumption is essential for Theorem 7.1. However, Theorem 7.1 is much harder than Theorem 6.1. We establish Theorem 7.1 by considering an arbitrary computation

$$\langle \sigma_0, \tau_0, T_0, \emptyset, \emptyset \rangle \longmapsto_o^* \langle \sigma', \tau', T', O', l' \rangle$$

where, unlike in the claim of Theorem 7.1, O' and l' may not be empty. For such a computation, we prove that there is a corresponding strong computation

$$\langle \sigma_0, \tau_0, T_0, \texttt{unit} \rangle \longmapsto_s^* \langle \sigma'', \tau'', T'', \texttt{unit} \rangle$$

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

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

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

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

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

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

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

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

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

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

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

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

$\langle \sigma, \tau, T, O, l\rangle \longmapsto_o \langle \sigma l, \tau, \mathit{origin}(O).T, \emptyset, \emptyset\rangle$ (Trans Undo)$_o$

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

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

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

$\langle \sigma, \tau, T.e.T', O, l\rangle \longmapsto_o \langle \sigma, \tau, T.T', (e, e, \emptyset, \emptyset).O, l\rangle$ (Trans Activate)$_o$

$\langle \sigma, \tau, T.\mathcal{U}[\ \texttt{protect}\ r\ ].T', O, l\rangle \longmapsto_o \langle \sigma, \tau[r \mapsto \texttt{P}], T.\mathcal{U}[\ r\ ].T', O, l\rangle$ (Trans DynPr)$_o$

$\langle \sigma, \tau, T.\mathcal{U}[\ \texttt{unprotect}\ r\ ].T', O, l\rangle \longmapsto_o \langle \sigma, \tau[r \mapsto \texttt{U}], T.\mathcal{U}[\ r\ ].T', O, l\rangle$ (Trans DynUn)$_o$
if $r \notin \mathit{dom}(l)$

Fig. 3. Transition rules with optimistic concurrency and dynamic separation.

where $\sigma' l'$ is an extension of σ'', τ' is an extension of τ'', and $T'' = T'.origin(O')$ up to reordering. Intuitively, this strong computation does not reflect the writes of the tries in O'; these writes are undone in $\sigma' l'$. (On the other hand, allocations are not undone.) In addition, in order to permit an inductive argument, we prove that $\tau'(r) = \mathtt{P}$ for every $r \in dom(l')$. We also prove that if O' has the form $O^\dagger.(e, f, a, P).O^{\dagger\dagger}$ and (e, f, a, P) does not conflict with $O^\dagger.O^{\dagger\dagger}$, then $\tau'(r) = \mathtt{P}$ for every $r \in a$. Note that $\tau'(r) = \mathtt{P}$ for $r \in dom(l')$ does not require an assumption about the absence of conflict but, in contrast, $\tau'(r) = \mathtt{P}$ for $r \in a$ does depend on such an assumption. Intuitively, this difference arises because we guarantee that transactions write only to protected reference locations whether or not there is a conflict, while zombies may read from unprotected reference locations after a conflict. Assuming the same absence of conflict, we also construct a further strong computation

$$\langle \sigma'', \tau'', T'', \mathtt{unit} \rangle \longmapsto_s \langle \sigma'', \tau'', T''', f \rangle \longmapsto_s^* \langle \sigma''', \tau''', T'''.P, e \rangle$$

where the first transition is an instance of (Trans Activate)$_s$ and

- $\sigma'(l' - a)$ is an extension of σ''',
- τ''' is an extension of τ'', and τ' is an extension of τ''',
- $T''' = T'.origin(O^\dagger.O^{\dagger\dagger})$ up to reordering,
- $\langle \sigma'', \tau'', T''', f \rangle \longmapsto_s^* \langle \sigma''', \tau''', T'''.P, e \rangle$ accesses the reference locations in a.

Intuitively, this further strong computation represents an execution that initially activates f and leads to e, corresponding to the try (e, f, a, P).

8 Conclusion

A notable aspect of our research on AME is that we have developed formal semantics alongside our software artifacts. The formal semantics have helped guide the practical implementation work and vice versa. As in the present study of dynamic separation, formal semantics shed light on the behavior of constructs and the properties of programming disciplines, even in the face of diverse implementation techniques.

Our objective is to enable the creation of programs by programmers with normal (not exceptional) skills, such that the programs will be satisfactory on current and future hardware, especially multi-processor and multi-core hardware. The programs must be semantically correct and must actually run correctly—at least the semantics and the implementations should be well-defined and simple enough that they are not an obstacle to correctness. The programs should also be efficient, so they should utilize concurrency where appropriate. Transactional memory with dynamic separation appears as a promising element in reconciling these goals.

Acknowledgements

This work was done at Microsoft Research. We are grateful to Katie Coons, Rebecca Isaacs, Yossi Levanoni, and Jean-Philippe Martin for helpful discussions and comments, and to Andrew Birrell, Johnson Hsieh, and Michael Isard for our joint work, which gave rise to the present paper.

References

[1] Martín Abadi, Andrew Birrell, Tim Harris, Johnson Hsieh, and Michael Isard. Dynamic separation for transactional memory. Technical Report MSR-TR-2008-43, Microsoft Research, March 2008.

[2] Martín Abadi, Andrew Birrell, Tim Harris, Johnson Hsieh, and Michael Isard. Implementation and use of transactional memory with dynamic separation. In Oege de Moor and Michael I. Schwartzbach, editors, *CC 2009: International Conference on Compiler Construction*, volume 5501 of *Lecture Notes in Computer Science*, pages 63–77. Springer-Verlag, March 2009.

[3] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL '08: Proc. 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 63–74, January 2008.

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

[5] Sarita V. Adve. *Designing memory consistency models for shared-memory multiprocessors*. PhD thesis, U. Wisconsin–Madison, 1993.

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

[7] 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.

[8] Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *WDDD '05: Proc. 4th Workshop on Duplicating, Deconstructing and Debunking*, pages 48–55, June 2005.

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

[10] Dave Dice and Nir Shavit. What really makes transactions faster? In *TRANSACT '06, 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[11] 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, October 2003.

[12] 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, June 2005.

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

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

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

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

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

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

[19] 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, October 2007.

[20] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. Technical Report 915, CS Dept, U. Rochester, February 2007.

[21] Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *CGO '07, International Symposium on Code Generation and Optimization*, pages 34–48, March 2007.

A Appendix: Proofs

Some Background Lemmas

In our proofs about static separation, we rely on results on AME, presented in `http://research.microsoft.com/en-us/projects/ame/ame-semantics-mar08.pdf`. Specifically, we rely on the following lemmas, which are numbered 4.1–4.5 in that document and which we restate here without proof.

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 $E\,;p \vdash \mathcal{E}[\ e'_0\ ] : t$.*

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

Lemma A.3 (Value substitution) *If $E, x : s, E'\,;p \vdash e : t$ and $E\,;q \vdash V : s$ then $E, E'\,;p \vdash e[V/x] : t$.*

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

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 of Lemma 5.1

Proof: We assume $\mathcal{DS}(\langle \sigma, \tau, T, \mathtt{unit}\rangle)$ and prove, by induction on the computation $\langle \sigma, \tau, T, \mathtt{unit}\rangle \longmapsto_s^* \langle \sigma', \tau', T', e'\rangle$, that $\langle \sigma', \tau', T', e'\rangle$ is locally good'. The assumption $\mathcal{DS}(\langle \sigma, \tau, T, \mathtt{unit}\rangle)$ implies that every state of the computation, including $\langle \sigma', \tau', T', e'\rangle$, is locally good.

Base Case: If $\langle \sigma, \tau, T, \mathtt{unit}\rangle$ takes zero steps, then we show that the state $\langle \sigma, \tau, T, \mathtt{unit}\rangle$ is locally good'. Since $\langle \sigma, \tau, T, \mathtt{unit}\rangle$ is locally good and its active expression is `unit`, it is also locally good'.

Induction Step: We assume that $\langle \sigma', \tau', T', e' \rangle$ is not locally good', in order to obtain a contradiction. If $\langle \sigma', \tau', T', e' \rangle$ is not locally good', then by definition one of the following statements must be true:

- $e' = \mathcal{P}[\ r := V\ ]$ or $e' = \mathcal{P}[\ !r\ ]$, and $\tau'(r) = \mathtt{U}$, or
- T' contains $\mathcal{U}[\ r := V\ ]$ or $\mathcal{U}[\ !r\ ]$, and $\tau'(r) = \mathtt{P}$.

We analyze these cases, focusing on writes; the treatment of reads is exactly analogous.

- If $e' = \mathcal{P}[\ r := V\ ]$ and $\tau'(r) = \mathtt{U}$, then $\langle \sigma', \tau', T', e' \rangle$ is not locally good—a contradiction.
- If T' contains $\mathcal{U}[\ r := V\ ]$ and $\tau'(r) = \mathtt{P}$, then $e' \neq \mathtt{unit}$ because $\langle \sigma', \tau', T', e' \rangle$ is locally good. Since the initial state of the computation is $\langle \sigma, \tau, T, \mathtt{unit} \rangle$ and $e' \neq \mathtt{unit}$, the computation must contain one or more applications of (Trans Activate)$_s$. We focus on the last transition that uses this rule. This transition activates some e'' that leads to e' in the following manner:

  $$\langle \sigma'', \tau'', T^\dagger.e''.T^{\dagger\dagger}, \mathtt{unit} \rangle \longmapsto_s \langle \sigma'', \tau'', T^\dagger.T^{\dagger\dagger}, e'' \rangle \longmapsto_s^* \langle \sigma', \tau', T', e' \rangle$$

  Because of the syntactic restriction that in any expression of the form `async` $(\ldots)$ all occurrences of `unprotected` are under a λ, the computation from e'' cannot spawn $\mathcal{U}[\ r := V\ ]$ via (Trans Async P)$_s$. Since T' contains $\mathcal{U}[\ r := V\ ]$, it must therefore be the case that $T^\dagger.T^{\dagger\dagger}$ contains $\mathcal{U}[\ r := V\ ]$. Moreover, τ' is an extension of τ'' (since the semantics allows only additions to the protection state when evaluating in a protected context), so $\tau''(r) = \tau'(r) = \mathtt{P}$. Therefore, $\langle \sigma'', \tau'', T^\dagger.e''.T^{\dagger\dagger}, \mathtt{unit} \rangle$ is not locally good—a contradiction again.

■

Proof of Theorem 5.2

Proof: This proof extends that of Theorem 6.1 of [3]. We prove that if $E \vdash \langle \sigma, e_1.\cdots.e_n, e \rangle$ and $\langle \sigma, \tau_E, e_1.\cdots.e_n, e \rangle \longmapsto_s \langle \sigma', \tau', e'_1.\cdots.e'_{n'}, e' \rangle$ then there is some E' such that $E' \vdash \langle \sigma', e'_1.\cdots.e'_{n'}, e' \rangle$ with $\tau' = \tau_{E'}$. The 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, $E' = E$, $\tau_E = \tau_{E'}$, 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 $\ldots$)$_s$:
  The typing of $\langle \sigma, T, \mathcal{P}[\ (\lambda x.\ e)\ V\ ] \rangle$ (or $\langle \sigma, T.\mathcal{U}[\ (\lambda x.\ e)\ V\ ].T', \texttt{unit} \rangle$) 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 $E\,; p_0 \vdash \lambda x.\ e : t_1 \rightarrow^{p_0} t_0$ and $E\,; p_0 \vdash V : t_1$ for some t_1, and therefore $E, x : t_1\,; p_0 \vdash e : t_0$. Lemma A.3 implies that $E\,; p_0 \vdash e[V/x] : t_0$. Moreover, Lemma A.1 yields a typing of $\langle \sigma, T, \mathcal{P}[\ e[V/x]\ ] \rangle$ (or $\langle \sigma, T.\mathcal{U}[\ e[V/x]\ ].T', \texttt{unit} \rangle$).
- (Trans Ref $\ldots$)$_s$:
  The typing of $\langle \sigma, T, \mathcal{P}[\ \texttt{ref}\ V\ ] \rangle$ (or $\langle \sigma, T.\mathcal{U}[\ \texttt{ref}\ V\ ].T', \texttt{unit} \rangle$) must rely on (Exp Ref). Specifically, we must have $E\,; p_0 \vdash \texttt{ref}\ V : \texttt{Ref}_{p_0}\ t_0$ for some t_0 and p_0, and therefore $E\,; p_0 \vdash V : t_0$. We extend E with $r : \texttt{Ref}_{p_0}\ t_0$. We can do this extension because $r \in \mathit{RefLoc} - \mathit{dom}(\sigma)$, hence $r \notin \mathit{dom}(E)$. After a weakening (adding $r : \texttt{Ref}_{p_0}\ t_0$ to E for typing $\langle \sigma, T, \mathcal{P}[\ \texttt{ref}\ V\ ] \rangle$) (or $\langle \sigma, T.\mathcal{U}[\ \texttt{ref}\ V\ ].T', \texttt{unit} \rangle$)), Lemma A.1 yields a typing of $\langle \sigma, T, \mathcal{P}[\ r\ ] \rangle$ (or $\langle \sigma, T.\mathcal{U}[\ r\ ].T', \texttt{unit} \rangle$). Because $r \in \mathit{dom}(E')$, and $E'(r) = \texttt{Ref}_{p_0}\ t_0$, we have that $\tau'(r) = \tau_{E'}(r)$, where $\tau' = \tau_E[r \mapsto p_0]$. For all other reference locations in τ', the desired result follows from the induction hypothesis.
- (Trans Deref $\ldots$)$_s$:
  The typing of $\langle \sigma, T, \mathcal{P}[\ !r\ ] \rangle$ (or $\langle \sigma, T.\mathcal{U}[\ !r\ ].T', \texttt{unit} \rangle$) 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 $E\,; p_0 \vdash r : \texttt{Ref}_{p_0}\ t_0$. Since r is a variable, its type must come from the environment E, so by hypothesis $E\,; p_0 \vdash V : t_0$ where $V = \sigma(r)$. Lemma A.1 yields a typing of $\langle \sigma, T, \mathcal{P}[\ V\ ] \rangle$ (or $\langle \sigma, T.\mathcal{U}[\ V\ ].T', \texttt{unit} \rangle$).
- (Trans Set $\ldots$)$_s$:
  The typing of $\langle \sigma, T, \mathcal{P}[\ r := V\ ] \rangle$ (or $\langle \sigma, T.\mathcal{U}[\ r := V\ ].T', \texttt{unit} \rangle$) 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 $E\,; p_0 \vdash r : \texttt{Ref}_{p_0}\ t_0$ for some p_0. Lemma A.1 allows us to transform a typing of $\langle \sigma, T, \mathcal{P}[\ r := V\ ] \rangle$ into a typing of $\langle \sigma, T, \mathcal{P}[\ \texttt{unit}\ ] \rangle$ (or a typing of $\langle \sigma, T.\mathcal{U}[\ r := V\ ].T', \texttt{unit} \rangle$ into a typing of $\langle \sigma, T.\mathcal{U}[\ \texttt{unit}\ ].T', \texttt{unit} \rangle$). Since $E\,; p_0 \vdash V : t_0$ and $E(r) = \texttt{Ref}_{p_0}\ t_0$, Lemma A.4 yields a typing of $\langle \sigma[r \mapsto V], T, \mathcal{P}[\ \texttt{unit}\ ] \rangle$ (or $\langle \sigma[r \mapsto V], T.\mathcal{U}[\ \texttt{unit}\ ].T', \texttt{unit} \rangle$).
- (Trans Async $\ldots$)$_s$:
  The typing of $\langle \sigma, T, \mathcal{P}[\ \texttt{async}\ e\ ] \rangle$ (or $\langle \sigma, T.\mathcal{U}[\ \texttt{async}\ e\ ].T', \texttt{unit} \rangle$) 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 $E\,; \texttt{P} \vdash e : \texttt{Unit}$. Lemma A.1 allows us to transform a typing of $\langle \sigma, T, \mathcal{P}[\ \texttt{async}\ e\ ] \rangle$ (or $\langle \sigma, T.\mathcal{U}[\ \texttt{async}\ e\ ].T', \texttt{unit} \rangle$) into a typing of $\mathcal{P}[\ \texttt{unit}\ ]$ (or $\mathcal{U}[\ \texttt{unit}\ ]$), and then into a typing of $\langle \sigma, e.T, \mathcal{P}[\ \texttt{unit}\ ] \rangle$ (or $\langle \sigma, e.T.\mathcal{U}[\ \texttt{unit}\ ].T', \texttt{unit} \rangle$) letting $n' = n + 1$.
- (Trans Block $\ldots$)$_s$:
  The typing of $\langle \sigma, T, \mathcal{P}[\ \texttt{blockUntil true}\ ] \rangle$ (or $\langle \sigma, T.\mathcal{U}[\ \texttt{blockUntil true}\ ].T', \texttt{unit} \rangle$) must rely on (Exp Block), specifically on a derivation of $E\,; p_0 \vdash \texttt{blockUntil true} : \texttt{Unit}$ for some p_0. Lemma A.1 yields a typing of $\langle \sigma, T, \mathcal{P}[\ \texttt{unit}\ ] \rangle$ (or $\langle \sigma, T.\mathcal{U}[\ \texttt{unit}\ ].T', \texttt{unit} \rangle$).
- (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. Lemma A.1 yields 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 $n' = n - 1$.
- (Trans DynUn)$_s$:
  By assumption, $\langle \sigma, T, e \rangle$ is well-typed. There are no typing rules for calls to `unprotect`, so this transition is not possible. This case is vacuous.
- (Trans DynPr)$_s$:
  By assumption, $\langle \sigma, T, e \rangle$ is well-typed. There are no typing rules for calls to `protect`, so this transition is not possible. This case is vacuous.

■

Proof of Theorem 5.3

Proof: We consider the computation:

$$\langle \sigma, \tau_E, T, e \rangle \longmapsto_s^* \langle \sigma', \tau', T', e' \rangle$$

Theorem 5.2 gives that there is some E' such that $E' \vdash \langle \sigma', T', e' \rangle$, and $\tau' = \tau_{E'}$. It suffices to show that the state $\langle \sigma', \tau_{E'}, T', e' \rangle$ is locally good. There are four cases:

- If $e' = \mathcal{P}[\ !r\ ]$, we need to show that $\tau_{E'}(r) = \texttt{P}$. We have some E' where $E' \vdash \langle \sigma', T', e' \rangle$. Lemma A.5 gives some t such that $E'\,; \texttt{P} \vdash\ !r : t$. Inversion on the typing rule (Exp Deref) gives that $E'\,; \texttt{P} \vdash r : \texttt{Ref}_\texttt{P}\ t$, and thus $E'(r) = \texttt{Ref}_\texttt{P}\ t$. Therefore $\tau_{E'}(r) = \texttt{P}$.
- If $T' = T^*.\mathcal{U}[\ !r\ ].T^{**}$, we need to show that $\tau_{E'}(r) = \texttt{U}$. We have some E' where $E' \vdash \langle \sigma', T', e' \rangle$. Lemma A.5 gives some t such that $E'\,; \texttt{U} \vdash\ !r : t$. Inversion on the typing rule (Exp Deref) gives that $E'\,; \texttt{U} \vdash r : \texttt{Ref}_\texttt{U}\ t$, and thus $E'(r) = \texttt{Ref}_\texttt{U}\ t$. Therefore $\tau_{E'}(r) = \texttt{U}$.
- If $e' = \mathcal{P}[\ r := V\ ]$, we need to show that $\tau_{E'}(r) = \texttt{P}$. We have some E' where $E' \vdash \langle \sigma', T', e' \rangle$. Lemma A.5 gives some t such that $E'\,; \texttt{P} \vdash r := V : t$. Inversion on the typing rule (Exp Set) gives that $E'\,; \texttt{P} \vdash r : \texttt{Ref}_\texttt{P}\ t$, and thus $E'(r) = \texttt{Ref}_\texttt{P}\ t$. Therefore $\tau_{E'}(r) = \texttt{P}$.
- If $T' = T^*.\mathcal{U}[\ r := V\ ].T^{**}$, we need to show that $\tau_{E'}(r) = \texttt{U}$. We have some E' where $E' \vdash \langle \sigma', T', e' \rangle$. Lemma A.5 gives some t such that $E'\,; \texttt{U} \vdash\ !r : t$. Inversion on the typing rule (Exp Set) gives that $E'\,; \texttt{U} \vdash r : \texttt{Ref}_\texttt{U}\ t$, and thus $E'(r) = \texttt{Ref}_\texttt{U}\ t$. Therefore $\tau_{E'}(r) = \texttt{U}$.

■

Proof of Lemma 5.5

Proof: We argue by contradiction. We assume that $\mathcal{DS}'(\langle\sigma, \tau, T, e\rangle)$, that $\langle\sigma, \tau, T, e\rangle \longmapsto_s^* \langle\sigma', \tau', T', e'\rangle$, and that $\langle\sigma', \tau', T', e'\rangle$ has a violation on a reference location r. We focus on the case where $\langle\sigma', \tau', T', e'\rangle$ has a write/write violation; the read/write case is analogous. Both of the following must be true:

- $e' = \mathcal{P}[\ r := V_1\ ]$, and
- $T' = T^*.\mathcal{U}[\ r := V_2\ ].T^{**}$.

Since $e' = \mathcal{P}[\ r := V_1\ ]$, $\tau'(r) = \texttt{P}$ by the definition of locally good'. Since $T' = T''.\mathcal{U}[\ r := V_2\ ].T'''$, $\tau'(r) = \texttt{U}$ by the definition of locally good'. Thus, we have that $\tau'(r) = \texttt{P}$ and $\tau'(r) = \texttt{U}$, simultaneously. This is a contradiction. ■

Proof of Theorem 6.1

Proof: The proof is by induction on the computation:

$$\langle\sigma_1, \sigma_2, \tau, T, e\rangle \longmapsto_t^* \langle\sigma_1', \sigma_2', \tau', T', e'\rangle$$

Base Case: If the two-heap computation took zero steps, then by assumption we have that $\langle\sigma, \tau, T, e\rangle$ corresponds to $\langle\sigma_1, \sigma_2, \tau, T, e\rangle$, with $dom(\sigma) = dom(\sigma_1) = dom(\sigma_2)$, and if $\tau(r) = \texttt{P}$, then $\sigma_1(r) = \sigma(r)$, and if $\tau(r) = \texttt{U}$, then $\sigma_2(r) = \sigma(r)$.

Induction Step: The induction step is an immediate consequence of Lemma A.6, below. ■

Lemma A.6 *Assume that $\mathcal{DS}(\langle\sigma, \tau, T, e\rangle)$, that $dom(\sigma) = dom(\sigma_1) = dom(\sigma_2)$, and that $\sigma_1(r) = \sigma(r)$ if $\tau(r) = \texttt{P}$ and $\sigma_2(r) = \sigma(r)$ if $\tau(r) = \texttt{U}$. Consider a transition with two heaps:*

$$\langle\sigma_1, \sigma_2, \tau, T, e\rangle \longmapsto_t \langle\sigma_1', \sigma_2', \tau', T', e'\rangle$$

Then there is a transition:

$$\langle\sigma, \tau, T, e\rangle \longmapsto_s \langle\sigma', \tau', T', e'\rangle$$

for some σ' such that $dom(\sigma') = dom(\sigma_1') = dom(\sigma_2')$ and, for every $r \in dom(\sigma')$, if $\tau'(r) = \texttt{P}$, then $\sigma_1'(r) = \sigma'(r)$, and if $\tau'(r) = \texttt{U}$, then $\sigma_2'(r) = \sigma'(r)$.

Proof: We consider a transition

$$\langle\sigma_1, \sigma_2, \tau, T, e\rangle \longmapsto_t \langle\sigma_1', \sigma_2', \tau', T', e'\rangle$$

and a state $\langle\sigma, \tau, T, e\rangle$ such that:

- $dom(\sigma) = dom(\sigma_1) = dom(\sigma_2)$, and
- for every $r \in dom(\sigma)$,
  · if $\tau(r) = \texttt{P}$, then $\sigma_1(r) = \sigma(r)$, and
  · if $\tau(r) = \texttt{U}$, then $\sigma_2(r) = \sigma(r)$,

in order to construct a state $\langle \sigma', \tau', T', e' \rangle$ and to establish a similar correspondence between this state and $\langle \sigma'_1, \sigma'_2, \tau', T', e' \rangle$.

The argument is by cases on the rule applied in the transition

$$\langle \sigma_1, \sigma_2, \tau, T, e \rangle \longmapsto_t \langle \sigma'_1, \sigma'_2, \tau', T', e' \rangle$$

- (Trans Appl ...)$_t$, (Trans Async ...)$_t$, (Trans Block ...)$_t$, (Trans Unprotect)$_t$, (Trans Close)$_t$, and (Trans Activate)$_t$:
  When $\langle \sigma_1, \sigma_2, \tau, T, e \rangle \longmapsto_t \langle \sigma'_1, \sigma'_2, \tau', T', e' \rangle$ by one of these rules, we have:
  · $\sigma'_1 = \sigma_1$,
  · $\sigma'_2 = \sigma_2$
  · $\tau' = \tau$.
  We simulate this transition with an application of (Trans Appl ...)$_s$, (Trans Async ...)$_s$, (Trans Block ...)$_s$, (Trans Unprotect)$_t$, (Trans Close)$_t$, and (Trans Activate)$_s$, respectively. This application gives the desired results for e' and T', where $\sigma' = \sigma$ and $\tau' = \tau$. By combining the new expressions for σ', τ', σ'_1, and σ'_2 with the hypothesis, we obtain the desired results.
- (Trans Ref P)$_t$:
  When $\langle \sigma_1, \sigma_2, \tau, T, e \rangle \longmapsto_t \langle \sigma'_1, \sigma'_2, \tau', T', e' \rangle$ with (Trans Ref P)$_t$, we have:
  · $T' = T$,
  · $e = \mathcal{P}[\ \texttt{ref}\ V\ ]$ yields $e' = \mathcal{P}[\ r\ ]$,
  · $\sigma'_1 = \sigma_1[r \mapsto V]$,
  · $\sigma'_2 = \sigma_2[r \mapsto V]$,
  · $\tau' = \tau[r \mapsto \texttt{P}]$.
  We simulate this transition with a corresponding application of (Trans Ref P)$_s$, to obtain:
  · $T' = T$,
  · $e' = \mathcal{P}[\ r\ ]$,
  · $\sigma' = \sigma[r \mapsto V]$,
  · $\tau' = \tau[r \mapsto \texttt{P}]$.
  In this case, it is sufficient to note that $dom(\sigma') = dom(\sigma'_1) = dom(\sigma'_2)$, and that $\sigma'(r) = \sigma'_1(r) = \sigma'_2(r)$.
- (Trans Deref P)$_t$:
  When $\langle \sigma_1, \sigma_2, \tau, T, e \rangle \longmapsto_t \langle \sigma'_1, \sigma'_2, \tau', T', e' \rangle$ with (Trans Deref P)$_t$, we have:
  · $T' = T$,
  · $e = \mathcal{P}[\ !r\ ]$ yields $e' = \mathcal{P}[\ \sigma_1(r)\ ]$,
  · $\sigma'_1 = \sigma_1$,
  · $\sigma'_2 = \sigma_2$,
  · $\tau' = \tau$.

By assumption, the strong computation obeys dynamic separation, thus $\tau(r) = \texttt{P}$ and (by the hypothesis) $\sigma_1(r) = \sigma(r)$. We simulate the transition with a corresponding application of (Trans Deref P)$_s$, to obtain:
· $T' = T$,
· $e' = \mathcal{P}[\ \sigma(r)\ ]$,
· $\sigma' = \sigma$,
· $\tau' = \tau$.
By combining these expressions with the hypothesis, we get $dom(\sigma') = dom(\sigma'_1) = dom(\sigma'_2)$; since $\tau'(r) = \texttt{P}$, it is sufficient to note that $\sigma'_1(r) = \sigma'(r)$.

- (Trans Set P)$_t$:
When $\langle \sigma_1, \sigma_2, \tau, T, e\rangle \longmapsto_t \langle \sigma'_1, \sigma'_2, \tau', T', e'\rangle$ with (Trans Set P)$_t$, we have:
· $T' = T$,
· $e = \mathcal{P}[\ r := V\ ]$ yields $e' = \mathcal{P}[\ \texttt{unit}\ ]$,
· $\sigma'_1 = \sigma_1[r \mapsto V]$,
· $\sigma'_2 = \sigma_2$,
· $\tau' = \tau$.
By assumption, the strong computation obeys dynamic separation, so $\tau(r) = \texttt{P}$. We simulate this transition with a corresponding application of (Trans Set P)$_s$, to obtain:
· $T' = T$,
· $e' = \mathcal{P}[\ \texttt{unit}\ ]$,
· $\sigma' = \sigma[r \mapsto V]$, and
· $\tau' = \tau$.
By combining these expressions with the hypothesis, we get $dom(\sigma') = dom(\sigma'_1) = dom(\sigma'_2)$; since $\tau'(r) = \texttt{P}$, it is sufficient to note that $\sigma'_1(r) = \sigma'(r)$.
- (Trans Ref U)$_t$:
When $\langle \sigma_1, \sigma_2, \tau, T, e\rangle \longmapsto_t \langle \sigma'_1, \sigma'_2, \tau', T', e'\rangle$ with (Trans Ref U)$_t$, we have:
· $T = T^*.\mathcal{U}[\ \texttt{ref}\ V\ ].T^{**}$ yields $T' = T^*.\mathcal{U}[\ r\ ].T^{**}$,
· $e = e' = \texttt{unit}$,
· $\sigma'_1 = \sigma_1[r \mapsto V]$,
· $\sigma'_2 = \sigma_2[r \mapsto V]$,
· $\tau' = \tau[r \mapsto \texttt{U}]$.
We simulate this transition with a corresponding application of (Trans Ref U)$_s$, to obtain:
· $T' = T^*.\mathcal{U}[\ r\ ].T^{**}$,
· $e' = \texttt{unit}$,
· $\sigma' = \sigma[r \mapsto V]$,
· $\tau' = \tau[r \mapsto \texttt{U}]$.
In this case, it is sufficient to note that $dom(\sigma') = dom(\sigma'_1) = dom(\sigma'_2)$, and that $\sigma'(r) = \sigma'_1(r) = \sigma'_2(r)$.
- (Trans Deref U)$_t$:
When $\langle \sigma_1, \sigma_2, \tau, T, e\rangle \longmapsto_t \langle \sigma'_1, \sigma'_2, \tau', T', e'\rangle$ with (Trans Deref U)$_t$, we have:
· $T = T^*.\mathcal{U}[\ !r\ ].T^{**}$ yields $T' = T^*.\mathcal{U}[\ \sigma_2(r)\ ].T^{**}$,

· $e = e' = \texttt{unit}$,
· $\sigma'_1 = \sigma_1$,
· $\sigma'_2 = \sigma_2$,
· $\tau' = \tau$.

By assumption, the strong computation obeys dynamic separation, thus $\tau(r) = \texttt{U}$ and (by the hypothesis) $\sigma_2(r) = \sigma(r)$. We simulate the transition with a corresponding application of (Trans Deref U)$_s$, to obtain:
· $T' = T^*.\mathcal{U}[\ \sigma(r)\ ].T^{**}$,
· $e' = \texttt{unit}$,
· $\sigma' = \sigma$,
· $\tau' = \tau$.

By combining these expressions with the hypothesis, we get $dom(\sigma') = dom(\sigma'_1) = dom(\sigma'_2)$; since $\tau'(r) = \texttt{U}$, it is sufficient to note that $\sigma'_2(r) = \sigma'(r)$.

- (Trans Set U)$_t$:
  When $\langle \sigma_1, \sigma_2, \tau, T, e\rangle \longmapsto_t \langle \sigma'_1, \sigma'_2, \tau', T', e'\rangle$ with (Trans Set U)$_t$, we have:
  · $T = T^*.\mathcal{U}[\ r := V\ ].T^{**}$ yields $T' = T^*.\mathcal{U}[\ \texttt{unit}\ ].T^{**}$,
  · $e = e' = \texttt{unit}$,
  · $\sigma'_1 = \sigma_1$,
  · $\sigma'_2 = \sigma_2[r \mapsto V]$,
  · $\tau' = \tau$.

  By assumption, the strong computation obeys dynamic separation so $\tau(r) = \texttt{U}$. We simulate the transition with a corresponding application of (Trans Set U)$_s$, to obtain:
  · $T' = T^*.\mathcal{U}[\ \texttt{unit}\ ].T^{**}$,
  · $e' = \texttt{unit}$,
  · $\sigma' = \sigma[r \mapsto V]$,
  · $\tau' = \tau$.

  By combining these expressions with the hypothesis, we get $dom(\sigma') = dom(\sigma'_1) = dom(\sigma'_2)$; since $\tau'(r) = \texttt{U}$, it is sufficient to note that $\sigma'_2(r) = \sigma'(r)$.
- (Trans DynPr)$_t$:
  When $\langle \sigma_1, \sigma_2, \tau, T, e\rangle \longmapsto_t \langle \sigma'_1, \sigma'_2, \tau', T', e'\rangle$ with (Trans DynPr)$_t$, we have:
  · $T = T^*.\mathcal{U}[\ \texttt{protect}\ r\ ].T^{**}$ yields $T' = T^*.\mathcal{U}[\ r\ ].T^{**}$,
  · $e = e' = \texttt{unit}$.

  There are two subcases.

(1) If $\tau(r) = \texttt{P}$, then
  · $\sigma'_1 = \sigma_1$,
  · $\sigma'_2 = \sigma_2$,
  · $\tau' = \tau$,
  · the hypothesis gives $\sigma_1(r) = \sigma(r)$.

(2) If $\tau(r) = \texttt{U}$, then
  · $\sigma'_1 = \sigma_1[r \mapsto \sigma_2(r)]$,
  · $\sigma'_2 = \sigma_2$,
  · $\tau' = \tau[r \mapsto \texttt{P}]$,

· the hypothesis gives $\sigma_2(r) = \sigma(r)$.

We simulate the transition with a corresponding application of (Trans DynPr)$_s$, to obtain:

· $T' = T^*.\mathcal{U}[\ r\ ].T^{**}$,

· $e = e' = \texttt{unit}$,

· $\sigma' = \sigma$,

· $\tau' = \tau[r \mapsto \texttt{P}]$.

By combining these expressions with the hypothesis, we get $dom(\sigma') = dom(\sigma'_1) = dom(\sigma'_2)$; since $\tau'(r) = \texttt{P}$, it is sufficient to note that $\sigma'_1(r) = \sigma'(r)$.

- (Trans DynUn)$_t$:

When $\langle \sigma_1, \sigma_2, \tau, T, e\rangle \longmapsto_t \langle \sigma'_1, \sigma'_2, \tau', T', e'\rangle$ with (Trans DynUn)$_t$, we have:

· $T = T^*.\mathcal{U}[\ \texttt{unprotect}\ r\ ].T^{**}$ yields $T' = T^*.\mathcal{U}[\ r\ ].T^{**}$,

· $e = e' = \texttt{unit}$.

There are two subcases.

(1) If $\tau(r) = \texttt{U}$, then

· $\sigma'_1 = \sigma_1$,

· $\sigma'_2 = \sigma_2$,

· $\tau' = \tau$,

· the hypothesis gives $\sigma_2(r) = \sigma(r)$.

(2) If $\tau(r) = \texttt{P}$, then

· $\sigma'_1 = \sigma_1$,

· $\sigma'_2 = \sigma_2[r \mapsto \sigma_1(r)]$,

· $\tau' = \tau[r \mapsto \texttt{U}]$,

· the hypothesis gives $\sigma_1(r) = \sigma(r)$.

We simulate the transition with a corresponding application of (Trans DynUn)$_s$, to obtain:

· $T' = T^*.\mathcal{U}[\ r\ ].T^{**}$,

· $e = e' = \texttt{unit}$,

· $\sigma' = \sigma$,

· $\tau' = \tau[r \mapsto \texttt{U}]$.

By combining these expressions with the hypothesis, we get $dom(\sigma') = dom(\sigma'_1) = dom(\sigma'_2)$; since $\tau'(r) = \texttt{U}$, it is sufficient to note that $\sigma'_2(r) = \sigma'(r)$.

■

Proof of Theorem 7.1

Proof: More generally, we consider a computation

$$\langle \sigma_0, \tau_0, T_0, \emptyset, \emptyset\rangle \longmapsto_o^* \langle \sigma', \tau', T', O', l'\rangle$$

and prove the following four facts:

(1) There is a strong computation $\langle \sigma_0, \tau_0, T_0, \texttt{unit} \rangle \longmapsto_s^* \langle \sigma'', \tau'', T'', \texttt{unit} \rangle$ where:
- $\sigma' l'$ is an extension of σ'',
- τ' is an extension of τ'',
- $T'' = T'.\mathit{origin}(O')$ 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 $\langle \sigma'', \tau'', T'', \texttt{unit} \rangle \longmapsto_s \langle \sigma'', \tau'', T''', f \rangle \longmapsto_s^* \langle \sigma''', \tau''', T'''.P, e \rangle$ where the first transition is an instance of (Trans Activate)$_s$ and
- $\sigma'(l' - a)$ is an extension of σ''',
- τ''' is an extension of τ'', and τ' is an extension of τ''',
- $T''' = T'.\mathit{origin}(O^\dagger.O^{\dagger\dagger})$ up to reordering,
- $\langle \sigma'', \tau'', T''', f \rangle \longmapsto_s^* \langle \sigma''', \tau''', T'''.P, e \rangle$ accesses (reads or writes) the reference locations in a.

(3) In addition, 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 for every $r \in a$ then $\tau'(r) = \texttt{P}$.

(4) For every $r \in \mathit{dom}(l')$, $\tau'(r) = \texttt{P}$.

The proof is by induction on the computation

$$\langle \sigma_0, \tau_0, T_0, \emptyset, \emptyset \rangle \longmapsto_o^* \langle \sigma', \tau', T', O', l' \rangle$$

Throughout, we work up to reorderings in the pool.

Base Case: Assume the low-level computation took zero steps:

$$\langle \sigma_0, \tau_0, T_0, \emptyset, \emptyset \rangle \longmapsto_o^* \langle \sigma_0, \tau_0, T_0, \emptyset, \emptyset \rangle$$

Then the strong computation also takes zero steps:

$$\langle \sigma_0, \tau_0, T_0, \texttt{unit} \rangle \longmapsto_s^* \langle \sigma_0, \tau_0, T_0, \texttt{unit} \rangle$$

The desired results are immediate.

Induction Step: The argument considers the last transition $\langle \sigma, \tau, T, O, l \rangle \longmapsto_o \langle \sigma', \tau', T', O', l' \rangle$ in the low-level semantics, with a case analysis on the rule applied in this transition.

- (Trans Appl P)$_o$, (Trans Async P)$_o$, (Trans Block P)$_o$:

(1) When $\langle \sigma, \tau, T, O, l \rangle \longmapsto_o \langle \sigma', \tau', T', O', l' \rangle$ by one of these rules, we have:
- $\sigma' = \sigma$,
- $\tau' = \tau$,
- $T' = T$,
- $\mathit{origin}(O') = \mathit{origin}(O)$,
- $l' = l$.

The induction hypothesis gives a strong computation ending in $\langle \sigma_s, \tau_s, T_s, \texttt{unit} \rangle$ such that:

· σl is an extension of σ_s,
· τ is an extension of τ_s,
· $T_s = T.origin(O)$ up to reordering.

The desired result is obtained by combining the new expressions with the induction hypothesis.

(2) There are two subcases:
   (a) If the further strong computation is of a try that did not transition in the low-level semantics, then the desired result follows from the induction hypothesis.
   (b) If the further strong computation is of the try that did transition in the low-level semantics using one of these rules, then we extend the strong computation with a corresponding application of (Trans Appl P)$_s$, (Trans Async P)$_s$, or (Trans Block P)$_s$, respectively.

(3) This transition does not modify a or τ. The desired result follows from the induction hypothesis.

(4) This transition does not modify l or τ. The desired result follows from the induction hypothesis.

• (Trans Ref P)$_o$:

(1) When $\langle \sigma, \tau, T, O, l \rangle \longmapsto_o \langle \sigma', \tau', T', O', l' \rangle$ by this rule, we have:
· $\sigma' = \sigma[r \mapsto V]$,
· $\tau' = \tau[r \mapsto \mathtt{P}]$,
· $T' = T$,
· $O = O^*.(\mathcal{P}[\ \mathtt{ref}\ V\ ], f, a, P).O^{**}$,
· $O' = O^*.(\mathcal{P}[\ r\ ], f, a, P).O^{**}$,
· $origin(O') = origin(O)$,
· $l' = l$.

The induction hypothesis gives a strong computation ending in $\langle \sigma_s, \tau_s, T_s, \mathtt{unit} \rangle$ where:
· σl is an extension of σ_s,
· τ is an extension of τ_s,
· $T_s = T.origin(O)$ up to reordering.

The desired result is obtained by combining the new expressions with the induction hypothesis.

(2) There are two subcases:
   (a) If the further strong computation is of a try that did not transition with (Trans Ref P)$_o$, then the desired result follows from the induction hypothesis.
   (b) If the further strong computation is of the try where (Trans Ref P)$_o$ operates, then we extend the strong computation with a corresponding application of (Trans Ref P)$_s$.

(3) This transition does not modify a, and τ' extends τ with r. The desired result follows from the induction hypothesis.

(4) In this case, $l' = l$ and τ' extends τ with r. The desired result follows from the induction hypothesis.

• (Trans Deref P)$_o$:

(1) When $\langle \sigma, \tau, T, O, l \rangle \longmapsto_o \langle \sigma', \tau', T', O', l' \rangle$ by this rule, we have:
- $\sigma' = \sigma$,
- $\tau' = \tau$,
- $T' = T$,
- $O = O^*.(\mathcal{P}[\ !r\ ], f, a, P).O^{**}$,
- $O' = O^*.(\mathcal{P}[\ \sigma(r)\ ], f, r.a, P).O^{**}$,
- $\mathit{origin}(O') = \mathit{origin}(O)$,
- $l' = l$.

The induction hypothesis gives a strong computation ending in $\langle \sigma_s, \tau_s, T_s, \texttt{unit} \rangle$ such that:
- σl is an extension of σ_s,
- τ is an extension of τ_s
- $T_s = T.\mathit{origin}(O)$ up to reordering.

The desired result is obtained by combining the new expressions with the induction hypothesis.

(2) There are two subcases:
   (a) If the further strong computation is of a try that did not transition with (Trans Deref P)$_o$, then the desired result follows from the induction hypothesis.
   (b) If the further strong computation is of the try where (Trans Deref P)$_o$ operates, then we extend the strong computation with a corresponding application of (Trans Deref P)$_s$.

(3) For the try that transitions in the low-level semantics, we have two subcases:
   (a) (e, f, a, P) conflicts as a result of this transition. In this case, the claim is vacuous.
   (b) (e, f, a, P) does not conflict. Claim (2) gives the further strong computation:

$$\langle \sigma'', \tau'', T''.f, \texttt{unit} \rangle \longmapsto_s \langle \sigma'', \tau'', T''', f \rangle \longmapsto_s^* \langle \sigma''', \tau''', T'''.P, e \rangle$$

   Since this computation ends with an application of (Trans Deref P)$_s$, it must be the case that the next-to-last active expression is of the form $\mathcal{P}[\ !r\ ]$. By the assumption that this computation obeys dynamic separation, it must be the case that $\tau'''(r) = \texttt{P}$, and because τ' extends τ''', it must also be the case that $\tau'(r) = \texttt{P}$. The induction hypothesis gives the desired result for all other locations in a.

   For other tries, Claim (3) follows by the induction hypothesis.

(4) In this case, $l' = l$ and $\tau' = \tau$. The desired result follows from the induction hypothesis.

• (Trans Set P)$_o$:

(1) When $\langle \sigma, \tau, T, O, l \rangle \longmapsto_o \langle \sigma', \tau', T', O', l' \rangle$ by this rule, we have:
- $\sigma' = \sigma[r \mapsto V]$,
- $\tau' = \tau$,
- $T' = T$,

· $O = O^*.(\mathcal{P}[\ r := V\ ], f, a, P).O^{**}$, and
$O' = O^*.(\mathcal{P}[\ \texttt{unit}\ ], f, r.a, P).O^{**}$,
· $origin(O') = origin(O)$,
· $r \in dom(l')$.

The induction hypothesis gives that there is a strong computation ending in $\langle \sigma_s, \tau_s, T_s, \texttt{unit} \rangle$ such that:

· σl is an extension of σ_s,
· τ is an extension of τ_s,
· $T_s = T.origin(O)$ up to reordering.

The desired result is obtained by combining the new expressions with the induction hypothesis.

(2) There are two subcases:
   (a) If the further strong computation is of a try that did not transition with (Trans Set P)$_o$, then the desired result follows from the induction hypothesis.
   (b) If the further strong computation is of the try where (Trans Set P)$_o$ operates, then we extend the strong computation with a corresponding application of (Trans Set P)$_s$.

(3) Examination of the low-level computation yields $r \in a$. To apply this transition, we require that $\tau(r) = \texttt{P}$. In this case, $\tau' = \tau$, so $\tau'(r) = \texttt{P}$. The induction hypothesis gives the desired result for all other locations in a.

(4) In this case, $r \in dom(l')$. By assumption, $\tau(r) = \texttt{P}$. Since $\tau' = \tau$, we know $\tau'(r) = \texttt{P}$. The induction hypothesis gives the desired result for all other locations in $dom(l')$.

• (Trans Appl U)$_o$, (Trans Block U)$_o$, (Trans Close)$_o$:

(1) When $\langle \sigma, \tau, T, O, l \rangle \longmapsto_o \langle \sigma', \tau', T', O', l' \rangle$ by one of these rules, we have:
· $\sigma' = \sigma$,
· $\tau' = \tau$,
· $T = T^*.\mathcal{U}[\ e\ ].T^{**}$,
· $T' = T^*.\mathcal{U}[\ e'\ ].T^{**}$,
· $O' = O$,
· $l' = l$.

The induction hypothesis gives a strong computation ending in $\langle \sigma_s, \tau_s, T_s, \texttt{unit} \rangle$ such that:

· σl is an extension of σ_s,
· τ is an extension of τ_s,
· $T_s = T^*.\mathcal{U}[\ e\ ].T^{**}.origin(O)$ up to reordering.

We extend the strong computation with a corresponding application of (Trans Appl U)$_s$, (Trans Block U)$_s$, and (Trans Close)$_s$, respectively, to obtain:

· $\sigma'' = \sigma_s$,
· $\tau'' = \tau_s$,
· $T'' = T^*.\mathcal{U}[\ e'\ ].T^{**}.origin(O)$ up to reordering.

The desired result is obtained by combining the new expressions with the induction hypothesis.

(2) The further strong computation follows from the induction hypothesis, with straightforward changes in the pool given by a corresponding application of (Trans Appl U)$_s$, (Trans Block U)$_s$, and (Trans Close)$_s$, respectively.
(3) The desired result follows from the induction hypothesis.
(4) The desired result follows from the induction hypothesis.

• (Trans Async U)$_o$:

(1) When $\langle \sigma, \tau, T, O, l\rangle \longmapsto_o \langle \sigma', \tau', T', O', l'\rangle$ by this rule, we have:
  · $\sigma' = \sigma$,
  · $\tau' = \tau$,
  · $T = T^*.\mathcal{U}[\ \texttt{async}\ e\ ].T^{**}$,
  · $T' = e.T^*.\mathcal{U}[\ \texttt{unit}\ ].T^{**}$,
  · $O' = O$,
  · $l' = l$.

  The induction hypothesis gives a strong computation ending in $\langle \sigma_s, \tau_s, T_s, \texttt{unit}\rangle$ such that:
  · σl is an extension of σ_s,
  · τ is an extension of τ_s,
  · $T_s = T^*.\mathcal{U}[\ \texttt{async}\ e\ ].T^{**}.\mathit{origin}(O)$ up to reordering.

  We extend the strong computation with a corresponding application of (Trans Async U)$_s$ to obtain:
  · $\sigma'' = \sigma_s$,
  · $\tau'' = \tau_s$,
  · $T'' = e.T^*.\mathcal{U}[\ \texttt{unit}\ ].T^{**}.\mathit{origin}(O)$ up to reordering.

  The desired result is obtained by combining the new expressions with the induction hypothesis.
(2) The further strong computation is given by the induction hypothesis, with straightforward changes in the pool given by a corresponding application of (Trans Async U)$_s$.
(3) The desired result follows from the induction hypothesis.
(4) The desired result follows from the induction hypothesis.

• (Trans Ref U)$_o$:

(1) When $\langle \sigma, \tau, T, O, l\rangle \longmapsto_o \langle \sigma', \tau', T', O', l'\rangle$ by this rule, we have:
  · $\sigma' = \sigma[r \mapsto V]$,
  · $\tau' = \tau[r \mapsto \texttt{U}]$,
  · $T = T^*.\mathcal{U}[\ \texttt{ref}\ V\ ].T^{**}$,
  · $T' = T^*.\mathcal{U}[\ r\ ].T^{**}$,
  · $O' = O$,
  · $l' = l$.

  The induction hypothesis gives a strong computation ending in $\langle \sigma_s, \tau_s, T_s, \texttt{unit}\rangle$ where:
  · σl is an extension of σ_s,
  · τ is an extension of τ_s,
  · $T_s = T^*.\mathcal{U}[\ \texttt{ref}\ V\ ].T^{**}.\mathit{origin}(O)$ up to reordering.

  We extend the strong computation with a corresponding application of

(Trans Ref U)$_s$ to obtain:
- $\sigma'' = \sigma_s[r \mapsto V]$,
- $\tau'' = \tau_s[r \mapsto \mathtt{U}]$,
- $T'' = T^*.\mathcal{U}[\ r\ ].T^{**}.\mathit{origin}(O)$ up to reordering.

This application gives the desired result for T''. We combine the new expressions for σ', l', τ', σ'', and τ'' with the induction hypothesis. Hence, $\sigma' l'$ extends σ'' and τ' extends τ''.

(2) The further strong computation is given by the induction hypothesis with the addition of some location, r, to σ and τ, and straightforward changes to the pool.

(3) Examination of the low-level computation yields that $r \notin a$ and τ' is an extension of τ with r. Thus, the desired result follows from the induction hypothesis.

(4) Examination of the low-level computation yields that $r \notin \mathit{dom}(l')$ and τ' is an extension of τ with r. The desired result follows from the induction hypothesis.

• (Trans Deref U)$_o$:

(1) When $\langle \sigma, \tau, T, O, l\rangle \longmapsto_o \langle \sigma', \tau', T', O', l'\rangle$ by this rule, we have:
- $\sigma' = \sigma$,
- $\tau' = \tau$,
- $T = T^*.\mathcal{U}[\ !r\ ].T^{**}$,
- $T' = T^*.\mathcal{U}[\ \sigma(r)\ ].T^{**}$,
- $O' = O$,
- $l' = l$.

The induction hypothesis gives a strong computation ending in $\langle \sigma_s, \tau_s, T_s, \mathtt{unit}\rangle$ such that:
- σl is an extension of σ_s,
- τ is an extension of τ_s,
- $T_s = T^*.\mathcal{U}[\ !r\ ].T^{**}.\mathit{origin}(O)$ up to reordering.

We extend the strong computation with a corresponding application of (Trans Deref U)$_s$ to obtain:
- $\sigma'' = \sigma_s$,
- $\tau'' = \tau_s$,
- $T'' = T^*.\mathcal{U}[\ \sigma_s(r)\ ].T^{**}.\mathit{origin}(O)$ up to reordering.

By assumption, the strong computation obeys dynamic separation, and thus $\tau_s(r) = \mathtt{U}$. Since τ extends τ_s, it must be that $\tau(r) = \mathtt{U}$. Claim (4) gives that $\tau(r') = \mathtt{P}$ for every $r' \in \mathit{dom}(l)$. Thus, it must be that $r \notin \mathit{dom}(l)$, so $\sigma l(r) = \sigma(r)$. Because σl extends σ_s, it must be the case that $\sigma(r) = \sigma_s(r)$.

(2) In this case, we must show that the location being read is not also read or written by the further strong computation. Thus, it suffices to know that $r \notin a$. By assumption, the strong computation that ends in $\langle \sigma'', \tau'', T'', \mathtt{unit}\rangle$ obeys dynamic separation, and the prior claim gives that $\tau''(r) = \mathtt{U}$. Also, the induction hypothesis gives that τ' extends τ''', and that τ''' extends τ''. Thus, $\tau'(r) = \mathtt{U}$. Claim (3) gives that for every $r' \in a$ then $\tau'(r') = \mathtt{P}$.

Thus, it must be that $r \notin a$, and therefore the further strong computation does not access r. Because the further strong computation neither reads nor writes to the reference location r, its execution commutes with the read operation.

(3) The desired result follows from the induction hypothesis.

(4) The desired result follows from the induction hypothesis.

- (Trans Set U)$_o$:

(1) When $\langle \sigma, \tau, T, O, l \rangle \longmapsto_o \langle \sigma', \tau', T', O', l' \rangle$ by this rule, we have:
   - $\sigma' = \sigma[r \mapsto V]$,
   - $\tau' = \tau$,
   - $T = T^*.\mathcal{U}[\ r := V\ ].T^{**}$,
   - $T' = T^*.\mathcal{U}[\ \texttt{unit}\ ].T^{**}$,
   - $O' = O$,
   - $l' = l$.

   The induction hypothesis gives a strong computation ending in $\langle \sigma_s, \tau_s, T_s, \texttt{unit} \rangle$ such that:
   - σl is an extension of σ_s,
   - τ is an extension of τ_s,
   - $T_s = T^*.\mathcal{U}[\ r := V\ ].T^{**}.\mathit{origin}(O)$ up to reordering.

   We extend the strong computation with a corresponding application of (Trans Set U)$_s$ to obtain:
   - $\sigma'' = \sigma_s[r \mapsto V]$,
   - $\tau'' = \tau_s$,
   - $T'' = T^*.\mathcal{U}[\ \texttt{unit}\ ].T^{**}.\mathit{origin}(O)$ up to reordering.

   By assumption, the strong computation obeys dynamic separation, and thus $\tau_s(r) = \texttt{U}$. Since $\tau'' = \tau_s$, and τ' extends τ'', it must be that $\tau'(r) = \texttt{U}$. Claim (4) gives that for every $r' \in \mathit{dom}(l)$, then $\tau'(r') = \texttt{P}$. Thus, $r \notin \mathit{dom}(l)$. In this case, $l' = l$, and therefore $r \notin \mathit{dom}(l')$. We combine the new expressions for σ', l', τ', σ'', and τ'' with the induction hypothesis. Hence, $\sigma' l'$ extends σ'' and τ' extends τ''.

(2) In this case, we must show that the location being modified is not also read or written by the further strong computation. Thus, it suffices to know that $r \notin a$. By assumption, the strong computation that ends in $\langle \sigma'', \tau'', T'', \texttt{unit} \rangle$ obeys dynamic separation, and the prior claim gives that $\tau''(r) = \texttt{U}$. Also, the induction hypothesis gives that τ' extends τ''', and that τ''' extends τ''. Thus, $\tau'(r) = \texttt{U}$. Claim (3) gives that for every $r' \in a$ then $\tau'(r') = \texttt{P}$. Thus, it must be that $r \notin a$ and because the further strong computation neither reads nor writes to the reference location r, its execution commutes with the write operation.

(3) This transition does not modify the access list of any try, and $\tau' = \tau$. The desired result follows from the induction hypothesis.

(4) Examination of the low-level computation yields that $l' = l$, and $\tau' = \tau$. The desired result follows from the induction hypothesis.

- (Trans Activate)$_o$:

(1) When $\langle \sigma, \tau, T, O, l \rangle \longmapsto_o \langle \sigma', \tau', T', O', l' \rangle$ by this rule, we have:

· $\sigma' = \sigma$,
· $\tau' = \tau$,
· $T = T^*.e.T^{**}$,
· $T' = T^*.T^{**}$,
· $O' = O.(e, e, \emptyset, \emptyset)$,
· $\mathit{origin}(O') = \mathit{origin}(O).e$,
· $l' = l$.

The induction hypothesis gives a strong computation ending in $\langle \sigma_s, \tau_s, T_s, \texttt{unit} \rangle$ such that:

· σl is an extension of σ_s,
· τ is an extension of τ_s,
· $T_s = T^*.e.T^{**}.\mathit{origin}(O)$ up to reordering.

We extend the strong computation with a corresponding application of (Trans Activate)$_s$ to obtain:

· $\sigma'' = \sigma_s$,
· $\tau'' = \tau_s$,
· $T'' = T^*.T^{**}.e.\mathit{origin}(O)$ up to reordering.

This application gives the desired result for T''. We combine the new expressions for σ', l', τ', σ'', and τ'' with the induction hypothesis. Hence, $\sigma' l'$ extends σ'' and τ' extends τ''.

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

(3) This transition does not add a location to the access list of any try, so for all tries in O, the desired result follows from the induction hypothesis. (Trans Activate)$_o$ creates a try where $a = \emptyset$, for which the desired result holds vacuously.

(4) This transition yields $l' = l$ and $\tau' = \tau$. The desired result follows from the induction hypothesis.

• (Trans DynPr)$_o$:

(1) When $\langle \sigma, \tau, T, O, l \rangle \longmapsto_o \langle \sigma', \tau', T', O', l' \rangle$ with this rule, we have:

· $\sigma' = \sigma$,
· $\tau' = \tau[r \mapsto \texttt{P}]$,
· $T = T^*.\mathcal{U}[\ \texttt{protect}\ r\ ].T^{**}$,
· $T' = T^*.T^{**}$,
· $O' = O$,
· $l' = l$.

The induction hypothesis gives a strong computation ending in $\langle \sigma_s, \tau_s, T_s, \texttt{unit} \rangle$ such that:

· σl is an extension of σ_s,
· τ is an extension of τ_s,
· $T_s = T^*.\mathcal{U}[\ \texttt{protect}\ r\ ].T^{**}.\mathit{origin}(O)$ up to reordering.

We extend the strong computation with a corresponding application of (Trans DynPr)$_s$ to obtain:

· $\sigma'' = \sigma_s$,

· $\tau'' = \tau_s[r \mapsto \texttt{P}]$,
· $T'' = T^*.\mathcal{U}[\ r\ ].T^{**}.origin(O)$ up to reordering.

This application gives the desired result for T''. We combine the new expressions for σ', l', τ', σ'', and τ'' with the induction hypothesis. Hence, $\sigma' l'$ extends σ'' and τ' extends τ''.

(2) The further strong computation is given by the induction hypothesis, with straightforward changes in the pool and the protection state that correspond to the application of $(\text{Trans DynPr})_s$.

(3) This transition updates the protection state of a reference location to `P`. The desired result follows from the induction hypothesis.

(4) This transition updates the protection state of a reference location to `P`. The desired result follows from the induction hypothesis.

• $(\text{Trans DynUn})_o$:

(1) When $\langle \sigma, \tau, T, O, l\rangle \longmapsto_o \langle \sigma', \tau', T', O', l'\rangle$ by this rule, we have:
· $\sigma' = \sigma$,
· $\tau' = \tau[r \mapsto \texttt{U}]$,
· $T = T^*.\mathcal{U}[\ \texttt{unprotect}\ r\ ].T^{**}$,
· $T' = T^*.T^{**}$,
· $O' = O$,
· $l' = l$,
· $r \notin dom(l)$.

The induction hypothesis gives a strong computation ending in $\langle \sigma_s, \tau_s, T_s, \texttt{unit}\rangle$ such that:
· σl is an extension of σ_s,
· τ is an extension of τ_s,
· $T_s = T^*.\mathcal{U}[\ \texttt{unprotect}\ r\ ].T^{**}.origin(O)$ up to reordering.

We extend the strong computation with a corresponding application of $(\text{Trans DynUn})_s$ to obtain:
· $\sigma'' = \sigma_s$,
· $\tau'' = \tau_s[r \mapsto \texttt{U}]$,
· $T'' = T^*.\mathcal{U}[\ r\ ].T^{**}.origin(O)$ up to reordering.

This application gives the desired result for T''. We combine the new expressions for σ', l', τ', σ'', and τ'' with the induction hypothesis. Hence, $\sigma' l'$ extends σ'' and τ' extends τ''.

(2) The further strong computation is given by the induction hypothesis, with a modification to the protection state of some reference location r.

(3) In the course of this new further strong computation, each reference location accessed must be in mode `P`, by dynamic separation. Since every reference location in a must be accessed, and since `protect` and `unprotect` do not work in protected code, we have that $\tau'''(r') = \texttt{P}$ for each $r' \in a$. Since τ' is an extension of τ''', we conclude that $\tau'(r') = \texttt{P}$ for each $r' \in a$.

(4) To make a transition with $(\text{Trans DynUn})_o$, we require that the reference location being unprotected is not in $dom(l)$. In this instance, $l' = l$, and thus the location is not in $dom(l')$. Thus, the desired result follows from the induction hypothesis.

• (Trans Undo)$_o$:

(1) When $\langle \sigma, \tau, T, O, l\rangle \longmapsto_o \langle \sigma', \tau', T', O', l'\rangle$ by this rule, we have:
- $\sigma' = \sigma l$,
- $\tau' = \tau$,
- $T' = \mathit{origin}(O).T$,
- $O' = \emptyset$,
- $l' = \emptyset$.

The induction hypothesis gives a strong computation ending in $\langle \sigma_s, \tau_s, T_s, \texttt{unit}\rangle$ such that:
- σl is an extension of σ_s,
- τ is an extension of τ_s,
- $T_s = T.\mathit{origin}(O)$ up to reordering.

or, in other words:
- σ' is an extension of σ_s,
- τ' is an extension of τ_s,
- $T_s = T'$ up to reordering.

The desired result follows from $O' = l' = \emptyset$.

(2) O' is $\emptyset$, so it does not have the form $O^\dagger.(e, f, a, P).O^{\dagger\dagger}$, hence this claim is vacuously true.

(3) Similarly, this claim is vacuously true as well.

(4) Since $l' = \emptyset$, $\tau'(r) = \texttt{P}$ for every $r \in \mathit{dom}(l')$, vacuously.

• (Trans Unprotect)$_o$:

(1) When $\langle \sigma, \tau, T, O, l\rangle \longmapsto_o \langle \sigma', \tau', T', O', l'\rangle$ by this rule, we have:
- $\sigma' = \sigma$,
- $\tau' = \tau$,
- $T' = T.\mathcal{P}[\ \texttt{unprotected}\ e\ ].P$,
- $O = O^*.(e, f, a, P).O^{**}$,
- $O' = O^*.O^{**}$,
- $\mathit{origin}(O') = \mathit{origin}(O^*.O^{**})$,
- $l' = l - a$.

By assumption, (e, f, a, P) does not conflict with $O^*.O^{**}$. The induction hypothesis gives that there is a strong computation ending in $\langle \sigma_s, \tau_s, T_s, \texttt{unit}\rangle$ such that:
- σl is an extension of σ_s,
- τ is an extension of τ_s,
- $T_s = T.\mathit{origin}(O').f$ up to reordering.

The induction hypothesis also gives a further strong computation:

$$\langle \sigma_s, \tau_s, T_s, \texttt{unit}\rangle \longmapsto_s \langle \sigma_s, \tau_s, T''', f\rangle \longmapsto_s^* \langle \sigma''', \tau''', T'''.P, \mathcal{P}[\ \texttt{unprotected}\ e\ ]\rangle$$

where:
- $\sigma'(l' - a)$ is an extension of σ''',
- τ''' is an extension of τ_s, and τ' is an extension of τ''',
- $T''' = T.\mathit{origin}(O')$.

We extend the strong computation with a corresponding application of

(Trans Unprotect)$_s$ to obtain:
  - $\sigma'' = \sigma'''$,
  - $\tau'' = \tau'''$,
  - $T'' = T.origin(O').P.\mathcal{P}[\ \texttt{unprotected}\ e\ ]$ up to reordering.

  By combining these new expressions with the induction hypothesis and making a note that $l' - a = l'$, we obtain:
  - $\sigma' l'$ extends σ'',
  - τ' extends τ'',
  - $T'' = T'.origin(O')$ up to reordering.

(2) Suppose that (e^*, f^*, a^*, P^*) is any other non-conflicting try. The further strong computation of (e^*, f^*, a^*, P^*) is 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 (Trans Unprotect)$_o$ implies that a^* and a do not intersect. Thus, the further strong computation that accesses only locations in a^* commutes with the computation that transitions with (Trans Unprotect)$_o$.

(3) This transition does not add a reference location to a, and $\tau' = \tau$. The desired result follows from the induction hypothesis.

(4) This transition does not add anything to the log, and $\tau' = \tau$. The transition with (Trans Unprotect)$_o$ may remove things from the log, so $dom(l') \subseteq dom(l)$. The desired result follows from the induction hypothesis.

- (Trans Done)$_o$: This case is almost identical to that of (Trans Unprotect)$_o$, but slightly simpler.

■