A Model of Dynamic Separation for Transactional Memory

Martín Abadi[1,2], Tim Harris[1], and Katherine F. Moore[1,3]

[1] Microsoft Research
[2] University of California, Santa Cruz
[3] 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 [14]. 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" [6], 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., [4]), 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"—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., [9, 2, 12]). 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 [2] (see also [8, 3, 5]). 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-transacted code rely on features not available for re-compilation (cf., e.g., [15]).

A companion paper [1] studies dynamic separation informally. That paper provides a more detailed design rationale, an instantiation for C#, and some conceptually easy but useful refinements, in particular for read-only data. It also discusses implementations, describing our working implementation (done in the context of Bartok-STM [10]) and a variant that serves as a debugging tool for testing whether a program obeys the dynamic-separation discipline. As a case study, it examines the use of dynamic separation in the context of a concurrent web-proxy application built over an asynchronous IO library.

The present paper focuses on the formal definition and study of dynamic separation. It provides a precise formulation of dynamic separation, in the setting of a small calculus with transactions (Sections 2–5). It also 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.

We present our results focusing on the Automatic Mutual Exclusion (AME) model [11, 2] (Section 2). However, as explained in our companion paper, our

approach applies also to other models for programming with transactions, for instance to TIC [16].

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.

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, defined 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-transacted code, AME provides block-structured `unprotected` sections. These must use existing mechanisms for synchronization. AME terminates the current atomic fragment before the code, and starts a new one afterwards.

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 syntax of the calculus, with the extensions required for dynamic separation.

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

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 (ef), allocation (`ref` e, which allocates a new reference location and returns it after initializing it to the value of e), dereferencing ($!e$, which returns the contents in the reference location that is the value of e), and assignment ($e := f$, which sets the reference location that is the value of e to the value of f). 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'.

We make a small technical restriction that does not affect the expressiveness of the calculus: 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 2, 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" [7], 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 (e.g., [2, 17]). 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-transacted code may take steps only when there is no current transaction taking steps [2]. We extend this strong semantics to the new constructs.

States. A state $\langle \sigma, \tau, T, e \rangle$ consists of a reference store σ, a protection state τ, a collection of expressions T, 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 write `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 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 $r \in RefLoc$ occurs in $\langle \sigma, \tau, T, e \rangle$, then $r \in dom(\sigma)$. We set:

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

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,

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

Fig. 1. Transition rules with dynamic separation.

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:

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

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.

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.

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 DynP)$_s$ and (Trans DynU)$_s$, 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 transactions. 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 S obeys the dynamic-separation discipline, and write $\mathcal{DS}(S)$, if whenever $S \longmapsto_s^* \langle \sigma, \tau, T, e \rangle$, the state $\langle \sigma, \tau, T, e \rangle$ is such that:

- 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 $\tau(r) = \texttt{U}$.

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 (transacted or not, respectively) of the operation, and if the same is true for any state reachable from S.

5.2 Comparison with Static Separation

Static separation can be defined as a type system; its details are straightforward, and for AME they are given in [2, 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. We obtain:

Theorem 1. *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.

5.3 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 dynamic separation.) 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. 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 1, as proof of our informal statement that dynamic separation is intermediate between violation-freedom and static separation.

Theorem 2. *If* $\mathcal{DS}(\langle \sigma, \tau, T, \texttt{unit} \rangle)$*, then* $\mathcal{VF}(\langle \sigma, \tau, T, \texttt{unit} \rangle)$*.*

Conversely, violation-freedom is not a sufficient condition for dynamic separation, for several 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. Strengthening violation-freedom so that it also constrains read-read concurrency, we have developed a method for taking a violation-free expression and adding calls to `protect` and `unprotect` so as to make it obey dynamic separation. We omit the details of our method, but briefly note its two main assumptions: (1) The method requires the absence of race conditions in unprotected computations, because race conditions could cause instrumentation to work incorrectly. (2) It also assumes that we can distinguish transacted and non-transacted 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 2. Transition rules with two heaps.

`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.

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. 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:

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

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 3. *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 [2], 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., [13]). 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.

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:

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

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 result of applying all elements of l to σ.

Many aspects of this semantics are explained in our previous work. Here we focus on the new ones, namely those related to dynamic separation.

Rule (Trans DynU)$_o$ 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 DynU)$_o$ can be satisfied

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

by performing an undo. However, an undo is never forced to happen. Indeed, the rules allow undo to happen at any point—possibly but not necessarily when there is a conflict. Conflict detection may be eager or lazy; the rules do not impose a particular strategy in this respect.

There is no corresponding subtlety in rule (Trans DynP)$_o$. 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. 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 4. *Assume that* $\mathcal{DS}(\langle \sigma, \tau, T, \mathtt{unit} \rangle)$. *Consider a computation:*

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

Then there is a computation:

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

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 3, the dynamic-separation assumption is essential for Theorem 4. However, Theorem 4 is much harder than Theorem 3.

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

2. M. Abadi, A. Birrell, T. Harris, and M. 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, 2008.
3. A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–37, 2006.
4. S. V. Adve. *Designing memory consistency models for shared-memory multiprocessors.* PhD thesis, U. Wisconsin–Madison, 1993.
5. E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt. The Fortress language specification, v1.0β. Technical report, Sun Microsystems, Mar. 2007.
6. C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *Proc. 2005 Workshop on Duplicating, Deconstructing and Debunking*, 2005.
7. D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.
8. T. Harris and K. Fraser. Language support for lightweight transactions. In *Proc. 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, 2003.
9. T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proc. 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60, 2005.
10. T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI '06: Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 14–25, 2006.
11. M. Isard and A. Birrell. Automatic mutual exclusion. In *Proc. 11th Workshop on Hot Topics in Operating Systems*, May 2007.
12. K. F. Moore and D. 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, 2008.
13. B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proc. Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–197, 2006.
14. N. Shavit and D. Touitou. Software transactional memory. In *Proc. 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, Aug. 1995.
15. T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in STM. In *PLDI '07: Proc. 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 78–88, 2007.
16. Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. Transactions with isolation and cooperation. In *Proc. 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 191–210, 2007.
17. M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory. Technical Report 915, CS Dept, U. Rochester, 2007.