

# High-Level Small-Step Operational Semantics for Transactions

Katherine F. Moore   Dan Grossman

University of Washington  
{kfm, djg}@cs.washington.edu

## Abstract

A high-level operational semantics for transactions can give transactions a precise meaning in the context of a programming language without exposing any details about how transactions are implemented. Such semantics give programmers a simple model and serve as a correctness criterion for transactional-memory implementations. This paper presents such a semantics for a simple  $\lambda$ -calculus with shared memory, threads, and transactions.

We use our approach to investigate two important aspects of a transactional programming language, namely spawning new threads (in particular, its interaction with transactions) and strong versus weak atomicity. For the former, we present a sound type system that ensures different ways of spawning threads are used correctly. For the latter, we use a different type system to prove the widely-held belief that if each mutable memory location is used outside transactions or inside transactions (but not both), then strong and weak atomicity are indistinguishable.

## 1. Introduction

A primary reason to add transactions to programming languages is that doing so makes it easier to reason about the behavior of shared-memory multithreaded programs. For such reasoning to be precise and well-founded, our languages need rigorous semantic definitions. Oversimplified, informal descriptions such as, “a transaction appears to happen all-at-once to other threads” are insufficient for deciding what programs mean, what programming idioms are correct, and what implementation strategies are meaning-preserving. Because *operational semantics*, in which languages are defined by judgments that reduce programs to simpler programs, is an expressive, convenient, rigorous, and well-understood approach, we advocate its use for defining transactions, resolving semantic subtleties, and proving properties of transactional languages. This paper presents our first results from pursuing this agenda.

Our goals are to formalize the simple “all-at-once” idea underlying transactions and prove properties about how this concept interacts with other language features. Therefore, our formal languages give a semantic definition to transactions at a very high level, essentially enforcing that a running

transaction has exclusive access to the shared-memory heap. That this basic approach is simple is a good thing; it corresponds exactly to why transactions simplify reasoning about programs. We do not want language definitions in terms of transactional-memory implementations. On the other hand, we do not simplify to the point that an entire transaction executes as one computational step since doing so abstracts away so much that it is not clear how to add interesting features such as internal parallelism (multiple threads within a transaction) or weak atomicity (nontransactional memory accesses that conflict with a transaction). As discussed in Section 5, this places our approach between prior work that was more suitable for proving implementations of transactions correct at the expense of simplicity [11, 22] and work that took the transactions-in-one-step approach that does not extend well to other features [8].

After presenting a basic language in the next section, we investigate two nontrivial extensions and prove appropriate results. (Full proofs are in a companion technical report [14].) First, we consider three possibilities for what it means to spawn a thread during a transaction and formalize all three in one language. A sound type-and-effect system classifies what code can run only in transactions, only outside transactions, or anywhere, thereby ensuring that no form of spawn occurs where it cannot execute successfully. Second, we present a language that lets code outside transactions access the heap while another thread executes a transaction. We prove that *if* all mutable memory is partitioned such that a location is accessed only outside or only inside transactions, then this weaker semantics is indistinguishable from the stronger one. While this result is not surprising, it is reassuring given the subtle pitfalls of nontransactional code bypassing transactional mechanisms [3, 12, 9, 20, 6, 21]. Moreover, we believe the structure of the proof will shed insight into how to prove that more sophisticated (and less obviously correct) invariants than a simple partition of memory are also correct.

## 2. Basic Formal Language

This section presents the syntax and small-step operational semantics for a  $\lambda$ -calculus with threads, shared memory, and transactions. It is primarily a starting point for the additions and type systems in the subsequent two sections, which

$$\begin{aligned}
e & ::= c \mid x \mid \lambda x.e \mid e_1 e_2 \mid \text{seq}(e_1, e_2) \\
& \quad \mid e_1 := e_2 \mid \text{ref } e \mid !e \\
& \quad \mid \text{spawn}_{\text{tl}} e \mid \text{atomic } e \\
& \quad \mid l \mid \text{inatomic}(e) \\
v & ::= c \mid \lambda x.e \mid l \\
H & ::= \cdot \mid H, l \mapsto v \\
T & ::= \cdot \mid T \parallel e \\
a & ::= \circ \mid \bullet
\end{aligned}$$

**Figure 1.** Base Syntax

investigate internal parallelism and weak atomicity. Three key design decisions characterize our approach:

- The semantics is *high-level*. It relies on implicit nondeterminism to find a successful execution sequence. There is no notion of transactions conflicting or aborting. Rather, a transaction is always isolated from other threads because no thread may access shared memory if another thread is in a transaction. This simple semantics provides a correctness criterion for more realistic implementations and a simple model for programmers.
- The semantics is *small-step*. In particular, transactions take many computational steps to complete. While this decision is an unnecessary complication for the basic language, it is essential for considering the additional thread interleavings that internal parallelism and weak atomicity introduce.
- The semantics is *strong*. Nontransactional code cannot observe or influence partially completed transactions. This lets us prove in Section 4 that strong-isolation semantics is equivalent to weak-isolation semantics under certain conditions. One cannot do such a proof without defining both semantics.

Our language does not have an explicit abort/retry. It is easy to add as in prior work [8] by simply having no evaluation rule for it: A transaction that aborts is simply one that can never be chosen by the nondeterministic semantics. However, it would complicate stating type-safety since we would have to accommodate an abort making a thread stuck.

## 2.1 Syntax

Figure 1 presents the formal abstract syntax for our small transactional language. Most expression forms are typical for a  $\lambda$ -calculus with mutable references, including constants ( $c$ ), variables ( $x$ ), functions ( $\lambda x.e$ ), function applications ( $e_1 e_2$ ), sequential composition ( $\text{seq}(e_1, e_2)$ ), memory allocation ( $\text{ref } e$ ), assignment ( $e_1 := e_2$ ), and dereference ( $!e$ ). Many omitted constructs, such as records, would be straightforward additions. In addition, we have thread-creation ( $\text{spawn}_{\text{tl}} e$ ), where the  $\text{tl}$  reminds us that threads must be created at top-level (not within a transaction), and atomic-blocks ( $\text{atomic } e$ ) for executing  $e$  as a transaction.

A program state has the form  $a; H; T$  where  $a$  indicates if any thread is currently executing a transaction ( $a = \bullet$  for yes and  $a = \circ$  for no),  $H$  is the mutable heap (a mapping from labels, also known as addresses, to values), and  $T$  is a collection of threads. Each thread is just an expression. We use  $T_1 \parallel T_2$  to combine two thread collections into a larger one, and we assume  $\parallel$  is commutative, associative, and has  $\cdot$  (the empty collection) as an identity. We write  $e$  in place of  $\cdot \parallel e$  where convenient.

At run-time we need two new expression forms, labels ( $l$ ) and  $\text{inatomic}(e)$ . The former represents a heap location. The latter represents a partially completed transaction with remaining computation  $e$ .

The program-state component  $a$  deserves additional discussion. Our *semantics* allows at most one thread to execute a transaction at a time. In essence  $a$  is like a “global lock” where  $\bullet$  indicates the lock is held. This is *not* to suggest our language is a desirable implementation, but it is the high-level semantics that enforces atomicity and isolation. We would like an efficient implementation to be *correct* if, by definition, it is equivalent to our semantics. (Correctness should also involve some notion of fairness, but we do not consider that in this work.)

## 2.2 Operational Semantics

Our small-step operational semantics rewrites one program state  $a; H; T$  to another  $a'; H'; T'$ . Source program  $e$  starts with  $\circ; \cdot; e$  and a terminal configuration has the form  $\circ; H; v_1 \parallel \dots \parallel v_n$ , i.e., all threads are values (and no transaction is active). Although the source program contains only a single  $e$ , the evaluation of  $e$  can spawn threads, which can spawn more threads, etc.

The rule PROGRAM chooses a thread nondeterministically and that thread takes a single step, which can affect  $a$  and  $H$  as well as possibly create a new thread. So the judgment form for single-thread evaluation is  $a; H; e \rightarrow a'; H'; e'; T$ , where  $T$  is  $\cdot$  if the step does not spawn a thread and some  $e''$  if it does.

For conciseness, we use evaluation contexts ( $E$ ) to identify where subexpressions are recursively evaluated and a single rule (CONTEXT) for propagating changes from evaluating the subexpression.<sup>1</sup> As usual, the inductive definition of  $E$  describes expressions with exactly one hole  $[\cdot]$  and  $E[e]$  means the expression resulting from replacing the hole in  $E$  with  $e$ . So, for example, we can use CONTEXT to derive  $a; H; \text{ref}(\text{seq}(e_1, e_2)) \rightarrow a'; H'; \text{ref}(\text{seq}(e'_1, e_2)); T$  if we can derive  $a; H; e_1 \rightarrow a'; H'; e'_1; T$ .

Rules for reducing sequences, memory allocations, and function calls are entirely conventional. In APPLY,  $e[v/x]$  means the capture-avoiding substitution of  $v$  for  $x$  in  $e$ .

The rules for reading and writing labels (SET and GET) require  $a = \circ$ , meaning no other thread is currently executing

<sup>1</sup> We do not treat the body of a transaction as an evaluation context precisely because we do not use the same  $a$  and  $a'$  for the subevaluation.

$$\boxed{a; H; e \rightarrow a'; H'; e'; T}$$

$E ::= [\cdot] \mid \text{seq}(E, e) \mid E := e \mid l := E \mid \text{ref } E \mid !E \mid E e \mid v E$

<p>CONTEXT</p> $\frac{a; H; e \rightarrow a'; H'; e'; T}{a; H; E[e] \rightarrow a'; H'; E[e']; T}$	<p>SEQ</p> $\frac{}{a; H; \text{seq}(v, e_2) \rightarrow a; H; e_2; \cdot}$	<p>APPLY</p> $\frac{}{a; H; (\lambda x. e) v_2 \rightarrow a; H; e[v_2/x]; \cdot}$
<p>ALLOC</p> $\frac{l \notin \text{Dom}(H)}{a; H; \text{ref } v \rightarrow a; H, l \mapsto v; l; \cdot}$	<p>SET</p> $\frac{}{\circ; H; l := v \rightarrow \circ; H, l \mapsto v; v; \cdot}$	<p>GET</p> $\frac{}{\circ; H; !l \rightarrow \circ; H; H(l); \cdot}$
<p>SPAWN TL</p> $\frac{}{a; H; \text{spawn}_{\text{tl}} e \rightarrow a; H; 0; e}$	<p>ENTER ATOMIC</p> $\frac{}{\circ; H; \text{atomic } e \rightarrow \bullet; H; \text{inatomic}(e); \cdot}$	<p>EXIT ATOMIC</p> $\frac{}{\bullet; H; \text{inatomic}(v) \rightarrow \circ; H; v; \cdot}$
<p>INATOMIC</p> $\frac{a; H; e \rightarrow a'; H'; e'; \cdot}{\bullet; H; \text{inatomic}(e) \rightarrow \bullet; H'; \text{inatomic}(e'); \cdot}$		
<p>PROGRAM</p> $\frac{a; H; e \rightarrow a'; H'; e'; T'}{a; H; T_A \parallel e \parallel T_B \rightarrow a'; H'; T_A \parallel e' \parallel T_B \parallel T'}$		

$$\boxed{a; H; T \rightarrow a'; H'; T'}$$

**Figure 2.** Evaluation Rules

a transaction. This encodes a high-level definition of strong isolation; it prohibits any memory conflict with a transaction. If no thread is in a transaction, then any thread may read and write the heap. We explain below how rule INATOMIC lets the computation within a transaction access the heap.

The rules defining how an expression enters or exits a transaction are of particular interest because they affect  $a$ . A thread can enter a transaction only if  $a = \circ$  (else ENTER ATOMIC does not apply), and it changes  $a$  to  $\bullet$ . Doing so prevents another thread from entering a transaction until EXIT ATOMIC (applicable only if the computation is finished, i.e., some value  $v$ ) changes  $a$  back to  $\circ$ .

A transaction itself must be able to access the heap (which, as discussed above, requires  $a = \circ$ ) and execute nested atomic expressions (which requires  $\circ$  for entry and  $\bullet$  for exit), but  $a$  is  $\bullet$  while a transaction is executing. That is why the hypothesis in rule INATOMIC allows any  $a$  and  $a'$  for the evaluation of the  $e$  inside the transaction. That way, the  $\bullet$  in the program state  $\bullet; H; \text{inatomic}(e)$  constrains only the *other* threads; the evaluation of  $e$  can choose any  $a$  and  $a'$  necessary to take a step. If we required  $a$  and  $a'$  to be  $\circ$ , then  $e$  could access the heap but it could not evaluate a nested atomic block. In the next section, internal parallelism requires a more general technique.

Note rule INATOMIC ensures a transaction does not spawn a thread (the hypothesis must produce thread-pool  $\cdot$ ), which

encodes that all spawns must occur at top-level. An expression like  $\text{inatomic}(\text{spawn}_{\text{tl}} e)$  is always stuck; there is no  $a$  and  $H$  with which it can take a step.

### 2.3 Type System

We could present a type system for this basic language, but most of the errors it would prevent are the standard ones (e.g., using an integer as a function). The only non-standard “stuck state” in our language so far occurs when a thread attempts to perform a spawn inside a transaction. The type-and-effect system presented in Section 3 prevents this error.

## 3. Internal Parallelism

While one reasonable semantics for spawn is that it is an error for it to occur in a transaction, there are several reasons to allow other possibilities. First, there is no conceptual problem with treating isolation and parallelism as orthogonal issues [16, 7, 11]. Second, if  $e$  spawns a thread (perhaps inside a library), then  $e$  and  $\text{atomic } e$  behave differently. Third, for some computations it may be sensible to delay any spawned threads until a transaction commits, and doing so is not difficult to implement. Fourth, it is undesirable to forfeit the performance benefits of parallelism every time we need to isolate a computation from some other threads.

This last issue becomes more important as the number of processors increases; otherwise transactions become a

$$\begin{aligned}
e & ::= \dots \mid \text{spawn}_{\text{oc}} e \mid \text{spawn}_{\text{ip}} e \\
& \quad \mid \text{inatomic}(a, e, T_{\text{oc}}, T_{\text{ip}}) \\
\tau & ::= \text{int} \mid \text{ref } \tau \mid \tau \xrightarrow{\varepsilon} \tau' \\
\varepsilon & ::= \text{tl} \mid \text{ip} \mid \emptyset \\
\Gamma & ::= \cdot \mid \Gamma, x : \tau
\end{aligned}$$

**Figure 3.** Internal Parallelism Syntax (extends Figure 1)

sequential bottleneck. For example, consider a concurrent hashtable with insert, lookup, and resize operations. Resize operations may be relatively rare and large yet still need to be isolated from other threads to avoid the complexities of concurrent operations. By parallelizing the resize operation while maintaining its isolation, we preserve correctness without allowing sequential resize operations to dominate performance.

In the rest of this section, we extend our formal language to show that different flavors of thread-spawn have reasonable semantics and a type-and-effect system can ensure the different flavors are used sensibly. Efficiently implementing parallelism within transactions is beyond the focus of our present work.

### 3.1 Syntax and Operational Semantics

Figure 3 presents the new syntax and Figure 4 presents the changes to the operational semantics.

The syntax additions are two new flavors of spawn expressions,  $\text{spawn}_{\text{oc}}$  (for “on commit”) and  $\text{spawn}_{\text{ip}}$  (for “internal parallelism”). The former is allowed to occur anywhere, but if it occurs inside a transaction, the spawned thread does not run until after the transaction commits. If the transaction aborts, the spawned thread never runs. The latter is allowed only within a transaction and the transaction does not commit until the spawned thread completes executing (i.e., becomes a value). One could certainly devise additional flavors of spawn; we believe these two new ones and  $\text{spawn}_{\text{tl}}$  cover a range of behaviors that are desirable in different situations. It is reasonable to provide them all in one programming language, perhaps with an undecorated spawn being a synonym for one of them. For example, the current Fortress specification [2] treats spawn as  $\text{spawn}_{\text{tl}}$ , but it also has constructs for fork-join style parallelism that our model could encode with  $\text{spawn}_{\text{ip}}$  inside a transaction.

The inatomic expression, which as before does not appear in source programs, has also changed. In addition to the  $e$  whose eventual result is the result of the transaction, it now carries an  $a$  and two threadpools,  $T_{\text{oc}}$  and  $T_{\text{ip}}$ . The  $a$  indicates whether  $e$  or any of the  $T_{\text{ip}}$  are currently executing a transaction. The  $T_{\text{oc}}$  are the threads that will be produced as “on commit” threads only when the transaction completes. The discussion of the semantics below explains inatomic further.

A single-thread evaluation step produces three possibly-empty threadpools  $T_{\text{tl}}$ ,  $T_{\text{oc}}$ , and  $T_{\text{ip}}$ . The evaluation rules

for the three flavors of spawn each put the new thread in the appropriate pool with the other pools empty. The CONTEXT rule propagates all three threadpools out to evaluation of the larger expression. Other rules, like those for assignment, sequences, function application, etc., only change by producing three empty pools instead of one. The rule for sequences has been included as an example.

The PROGRAM rule requires that the thread chosen for evaluation produces an empty  $T_{\text{ip}}$ , whereas  $T_{\text{tl}}$  and  $T_{\text{oc}}$  are added to the global pool of threads. In other words, it is an error to create an “internally parallel” thread outside a transaction and other flavors of threads are spawned immediately.

In a manner similar to the base language, entering a transaction changes  $\circ$  to  $\bullet$ , does not change the heap and creates no threads. The resulting expression  $\text{inatomic}(\circ, e, \cdot, \cdot)$  is a transaction with no nested transaction (hence the  $\circ$ ), no delayed threads (the first  $\cdot$ ) and no internally parallel threads (the second  $\cdot$ ).

For a transaction  $\text{inatomic}(a, e, T_{\text{oc}}, T_{\text{ip}})$ , either  $e$  or any thread in  $T_{\text{ip}}$  can take a step, using INATOMIC DISTINGUISHED or INATOMIC PARALLEL, respectively. The only reason to distinguish  $e$  is so inatomic produces a result; in languages where the body is a statement that produces no result we could combine these two rules. In either case, we recursively evaluate the thread using  $a$  and producing an  $a'$ ,  $H'$ ,  $e'$ ,  $T'_{\text{oc}}$ , and  $T'_{\text{ip}}$ . As in the base language, evaluation inside a transaction may not spawn a toplevel thread. The  $a'$ ,  $T'_{\text{oc}}$ , and  $T'_{\text{ip}}$  are all added to the resulting expression, i.e., they are part of the new state of the transaction. In particular, note that parallel threads in the transaction may produce other parallel or on-commit threads. Any heap changes are propagated outward immediately, which is no problem because the outer state is  $\bullet$ .

A transaction is complete when the distinguished expression and all parallel threads are values. Rule EXIT ATOMIC then propagates out all the on-commit threads in a single step. Notice that a transaction never produces any threads visible outside the transaction until this point.

Unlike in the base language, nested transactions are important; they allow a thread inside a transaction to perform multiple heap accesses atomically with respect to other threads in the transaction. At each level of nesting, we have an explicit  $a$  to ensure at most one of the threads is in a transaction and (because we still have strong isolation) if a thread is in a transaction, then no other threads access the heap. However, in the most-nested transaction, parallel threads may access the heap simultaneously. Finally, note that on-commit threads spawned inside nested transactions do not run until the outermost transaction commits. Other possibilities exist, but the soundness of our type-and-effect system depends on this choice.

### 3.2 Type System

The language just presented has several error states. These include common type errors (e.g., treating an integer as a

$$\boxed{a; H; e \rightarrow a'; H'; e'; T_{tl}; T_{oc}; T_{ip}}$$

$$\begin{array}{c}
\text{CONTEXT} \\
\frac{a; H; e \rightarrow a'; H'; e'; T_{tl}; T_{oc}; T_{ip}}{a; H; E[e] \rightarrow a'; H'; E[e']; T_{tl}; T_{oc}; T_{ip}}
\end{array}
\qquad
\begin{array}{c}
\text{SEQ} \\
\frac{}{a; H; \text{seq}(v, e_2) \rightarrow a; H; e_2; \cdot; \cdot}
\end{array}$$

$$\begin{array}{c}
\text{SPAWN TL} \\
\frac{}{a; H; \text{spawn}_{tl} e \rightarrow a; H; 0; e; \cdot; \cdot}
\end{array}
\qquad
\begin{array}{c}
\text{SPAWN OC} \\
\frac{}{a; H; \text{spawn}_{oc} e \rightarrow a; H; 0; \cdot; e; \cdot}
\end{array}
\qquad
\begin{array}{c}
\text{SPAWN IP} \\
\frac{}{a; H; \text{spawn}_{ip} e \rightarrow a; H; 0; \cdot; \cdot; e}
\end{array}$$

$$\begin{array}{c}
\text{ENTER ATOMIC} \\
\frac{}{\circ; H; \text{atomic } e \rightarrow \bullet; H; \text{inatomic}(\circ, e, \cdot, \cdot); \cdot; \cdot}
\end{array}
\qquad
\begin{array}{c}
\text{EXIT ATOMIC} \\
\frac{}{\bullet; H; \text{inatomic}(\circ, v, T_{oc}, (v_1 \parallel \dots \parallel v_n)) \rightarrow \circ; H; v; \cdot; T_{oc}; \cdot}
\end{array}$$

$$\begin{array}{c}
\text{INATOMIC DISTINGUISHED} \\
\frac{a; H; e \rightarrow a'; H'; e'; \cdot; T'_{oc}; T'_{ip}}{\bullet; H; \text{inatomic}(a, e, T_{oc}, T_{ip}) \rightarrow \bullet; H'; \text{inatomic}(a', e', (T_{oc} \parallel T'_{oc}), (T_{ip} \parallel T'_{ip})); \cdot; \cdot}
\end{array}$$

$$\begin{array}{c}
\text{INATOMIC PARALLEL} \\
\frac{a; H; e \rightarrow a'; H'; e'; \cdot; T'_{oc}; T'_{ip}}{\bullet; H; \text{inatomic}(a, e_0, T_{oc}, (T_{ip} \parallel e \parallel T''_{ip})) \rightarrow \bullet; H'; \text{inatomic}(a', e_0, (T_{oc} \parallel T'_{oc}), ((T_{ip} \parallel e' \parallel T''_{ip}) \parallel T'_{ip})); \cdot; \cdot}
\end{array}$$

$$\boxed{a; H; T \rightarrow a'; H'; T'}$$

$$\begin{array}{c}
\text{PROGRAM} \\
\frac{a; H; e \rightarrow a'; H'; e'; T_{tl}; T_{oc}; \cdot}{a; H; T_A \parallel e \parallel T_B \rightarrow a'; H'; T_A \parallel e' \parallel T_B \parallel T_{tl} \parallel T_{oc}}
\end{array}$$

**Figure 4.** Internally Parallel Evaluation Rules (selected rules omitted)

function), performing a top-level spawn inside a transaction, and performing an internally parallel spawn outside a transaction. We now present a type-and-effect system that soundly and conservatively prohibits such errors (Figure 5). For proving type safety, Section 3.3 extends the type system to run-time states, including expressions with labels and inatomic subexpressions.

The typing judgment  $\Gamma; \varepsilon \vdash e : \tau$  means that (1)  $e$  has type  $\tau$  using  $\Gamma$  to determine the types of variables, and (2) executing  $e$  only spawns threads of the flavors allowed by  $\varepsilon$ . A source program  $e$  type-checks if we can derive  $\cdot; tl \vdash e : \tau$  for some  $\tau$ . Since (1) is completely standard, we focus on (2), which is what makes our judgment an effect system.

The effect  $\emptyset$  describes computations that do not spawn top-level or internally-parallel threads. It allows on-commit threads; creating them never leads to dynamic errors. Similarly, effect  $tl$  permits on-commit and top-level threads, and  $ip$  permits on-commit and internally-parallel threads. We do not have an effect that allows all three flavors of spawn. Adding this effect is sound but not useful because code that type-checked only under this most-permissive effect could run safely neither inside *nor* outside a transaction.

Most aspects of our effect system are standard. Expressions that do not spawn threads can type-check with any effect. Values and variables are examples, so, for example, T-CONST allows any  $\varepsilon$ . By not requiring effect  $\emptyset$  in rules

like T-CONST, rules like T-SEQ and T-SET can use the same effect for both subexpressions. For example, we can derive  $x:\text{ref int}; tl \vdash (x := 17; \text{spawn}_{tl} 42) : \text{int}$ .<sup>2</sup> As usual, functions have *latent effects*, meaning function types carry an effect that occurs when the function is called. A function itself can have any effect, but the effect of its body is included in the effect of any call to it (see T-LAMBDA and T-APP). In T-APP, the simple subeffect relation allows using a function with latent effect  $\emptyset$  in a computation with effect  $tl$  or  $ip$ . In practice, we expect most functions type-check under effect  $\emptyset$ , which implies they definitely do not spawn threads. This subeffecting allows such functions to be called anywhere.

The most interesting rules are for atomic-blocks and spawn expressions. The body of an atomic-block must type-check under  $ip$ , but the atomic-block itself is allowed anywhere (thus enabling nested transactions and functions containing atomic-blocks that can be called inside and outside transactions). Because all  $\text{spawn}_{tl}$  expressions must execute outside transactions, the effect of the spawn *and* of the inner expression must be  $tl$ . By contrast, all expressions *created by*  $\text{spawn}_{oc}$  are evaluated at the top level (requiring effect  $tl$ ), but it is acceptable to execute the spawn expression itself at top-level or inside a transaction. Therefore, like for atomic-blocks, we allow the unconstrained effect  $\varepsilon$  for  $\text{spawn}_{oc}$ . Finally,  $\text{spawn}_{ip}$  requires effect  $ip$  for the entire expression and

<sup>2</sup>A fine alternative is adding an effect-subsumption rule.

$$\boxed{\varepsilon \leq \varepsilon'}$$

$$\text{REFLEXIVE} \\ \frac{}{\varepsilon \leq \varepsilon}$$

$$\text{EMPTY} \\ \frac{}{\emptyset \leq \varepsilon}$$

$$\boxed{\Gamma; \varepsilon \vdash e : \tau}$$

$$\begin{array}{c} \text{T-CONST} \\ \frac{}{\Gamma; \varepsilon \vdash c : \text{int}} \end{array} \quad \begin{array}{c} \text{T-VAR} \\ \frac{}{\Gamma; \varepsilon \vdash x : \Gamma(x)} \end{array} \quad \begin{array}{c} \text{T-SEQ} \\ \frac{\Gamma; \varepsilon \vdash e_1 : \tau_1 \quad \Gamma; \varepsilon \vdash e_2 : \tau_2}{\Gamma; \varepsilon \vdash \text{seq}(e_1, e_2) : \tau_2} \end{array} \quad \begin{array}{c} \text{T-SET} \\ \frac{\Gamma; \varepsilon \vdash e_1 : \text{ref } \tau \quad \Gamma; \varepsilon \vdash e_2 : \tau}{\Gamma; \varepsilon \vdash e_1 := e_2 : \tau} \end{array}$$

$$\begin{array}{c} \text{T-REF} \\ \frac{\Gamma; \varepsilon \vdash e : \tau}{\Gamma; \varepsilon \vdash \text{ref } e : \text{ref } \tau} \end{array} \quad \begin{array}{c} \text{T-GET} \\ \frac{}{\Gamma; \varepsilon \vdash !e : \tau} \end{array} \quad \begin{array}{c} \text{T-LAMBDA} \\ \frac{\Gamma, x : \tau_1; \varepsilon' \vdash e : \tau_2}{\Gamma; \varepsilon \vdash \lambda x. e : \tau_1 \xrightarrow{\varepsilon'} \tau_2} \end{array} \quad \begin{array}{c} \text{T-APP} \\ \frac{\Gamma; \varepsilon \vdash e_1 : \tau_1 \xrightarrow{\varepsilon'} \tau_2 \quad \Gamma; \varepsilon \vdash e_2 : \tau_1 \quad \varepsilon' \leq \varepsilon}{\Gamma; \varepsilon \vdash e_1 e_2 : \tau_2} \end{array}$$

$$\begin{array}{c} \text{T-ATOMIC} \\ \frac{\Gamma; \text{ip} \vdash e : \tau}{\Gamma; \varepsilon \vdash \text{atomic } e : \tau} \end{array} \quad \begin{array}{c} \text{T-SPAWN-TL} \\ \frac{}{\Gamma; \text{tl} \vdash e : \tau} \end{array} \quad \begin{array}{c} \text{T-SPAWN-OC} \\ \frac{}{\Gamma; \text{tl} \vdash e : \tau} \end{array} \quad \begin{array}{c} \text{T-SPAWN-IP} \\ \frac{}{\Gamma; \text{ip} \vdash e : \tau} \end{array}$$

**Figure 5.** Types and Effects for Source Programs

the spawned expression because both execute only within a transaction.

Note that if our language had expressions other than  $\text{spawn}_{\text{tl}}$  that could not occur in transactions (e.g., irreversible I/O), our effect system could statically prevent such errors in the same way.

### 3.3 Type Safety

Type safety means our computation never gets stuck. However, this does not mean that every thread can always proceed. For example, when one thread is in a transaction, another thread is (temporarily) stuck if its next step is to enter a transaction. Therefore, our type-safety theorem claims only that some thread (at least one, possibly many) can always proceed unless we have properly terminated:

**Theorem 3.1 Type Safety** *If (1)  $\text{tl} \vdash e : \tau$ , (2) after some number of steps  $\circ; \cdot; e$  becomes  $a; H; T$ , and (3) not all threads in  $T$  are values, then there exists a thread  $e$  in  $T$  such that  $a; H; e$  can take a single-thread evaluation step.*

As usual, we prove this theorem by showing preservation (any evaluation step from a well-typed state produces a well-typed state) and progress (no well-typed state is stuck) [25]. Doing so requires extending the type system to full run-time states  $a; H; T$  including the expression forms that do not appear in source programs. This extended system, presented in Figure 6, is only for the purpose of the proof. Proof details are in an available technical report [14]; here we sketch the interesting invariants that carefully completing a rigorous proof reveals.

To set the stage, most of the extensions are straightforward. To prove a state is well-typed ( $\vdash a; H; T$ ), we need

(1) the heap is well-typed ( $\Gamma \vdash H : \Gamma$ ),<sup>3</sup> and (2) each thread type-checks under effect  $\text{tl}$  and the labels in the heap ( $\Gamma; \text{tl} \vdash T$ ). Note our definition of  $\Gamma$  now includes types for labels. Also note when type-checking the heap effects are irrelevant because the heap contains only values and values never spawn threads. The third obligation for a well-typed state —  $\text{correct}(a, T)$  — is discussed below.

The typing rule for labels is as expected. The typing rule for functions has the new hypothesis  $\text{not-active}(e)$ . This is a technical point ensuring that a function body never contains a partially completed transaction. While this is true for any state resulting from a source program, it is an invariant that we must establish holds during evaluation. Otherwise, a function call could lead to a state where two threads were executing transactions simultaneously. Formal syntax-directed rules for  $\text{not-active}(e)$  are in the technical report, but as Figure 7 describes, they simply encode that no inatomic expression occurs in  $e$ .

The typing rule for evaluation inside a transaction has several subtleties. Because  $e$  and  $T_{\text{ip}}$  evaluate within a transaction, they must have effect  $\text{ip}$ . Similarly,  $T_{\text{oc}}$  is not evaluated until the top level so it must have effect  $\text{tl}$ . As with atomic expressions, the overall effect of inatomic is unconstrained to allow nesting. As with function bodies, the not-yet-running threads  $T_{\text{oc}}$  must not have inatomic subexpressions.

All that remains is to explain the hypotheses  $\text{correct}(a, T)$  and  $\text{correct}(a, e \parallel T)$  for typing program states and inatomic expressions, respectively. These judgments, defined formally in the technical report and summarized in Figure 7, are essential for showing that each  $a$  is correct —  $a = \bullet$  if

<sup>3</sup> Using  $\Gamma$  twice is a technical trick that allows cycles in the heap.

$\Gamma; \varepsilon \vdash e : \tau$ , additions and changes

$\Gamma ::= \dots \mid \Gamma, l : \tau$

$$\frac{\text{T-LABEL} \quad \Gamma(l) = \tau}{\Gamma; \varepsilon \vdash l : \text{ref } \tau}$$

$$\frac{\text{T-LAMBDA} \quad \Gamma, x : \tau; \varepsilon' \vdash e : \tau_2 \quad \text{not-active}(e)}{\Gamma; \varepsilon \vdash \lambda x. e : \tau_1 \xrightarrow{e'} \tau_2}$$

$$\frac{\text{T-INATOMIC} \quad \Gamma; \text{ip} \vdash e : \tau \quad \Gamma; \text{tl} \vdash T_{oc} \quad \Gamma; \text{ip} \vdash T_{ip} \quad \text{not-active}(T_{oc}) \quad \text{correct}(a, e \parallel T_{ip})}{\Gamma; \varepsilon \vdash \text{inatomic}(a, e, T_{oc}, T_{ip}) : \tau}$$

$\Gamma \vdash H : \Gamma'$

$\Gamma; \varepsilon \vdash T$

$\vdash a; H; T$

$$\frac{}{\Gamma \vdash \cdot : \cdot} \quad \frac{\Gamma \vdash H : \Gamma' \quad \Gamma; \varepsilon \vdash v : \tau}{\Gamma \vdash H, l \mapsto v : \Gamma', l : \tau} \quad \frac{}{\Gamma; \varepsilon \vdash \cdot} \quad \frac{\Gamma; \varepsilon \vdash T \quad \Gamma; \varepsilon \vdash e : \tau}{\Gamma; \varepsilon \vdash T \parallel e} \quad \frac{\Gamma \vdash H : \Gamma \quad \Gamma; \text{tl} \vdash T \quad \text{correct}(a, T)}{\vdash a; H; T}$$

**Figure 6.** Extensions for typechecking internally parallel program state. (See also Figure 7.)

not-active( $e$ ) not-active( $T$ )	$e$ (or $T$ ) contains no inatomic expression.
active( $e$ ) active( $T$ )	$e$ (or $T$ ) contains exactly 1 non-nested inatomic expression, and that occurrence is in a “sensible” syntactic position. (See discussion for more detail.)
correct( $a, T$ )	( $a = \circ$ and not-active( $T$ )) or ( $a = \bullet$ and active( $T$ ))

**Figure 7.** Active, Not-active, and Correct Atomicity

and only if exactly one thread is in a transaction, and  $a = \circ$  if and only if no thread is in a transaction. If this invariant does not hold, then the machine can be stuck. For example, if  $a = \bullet$ , no thread is in a transaction, and every thread is blocked waiting to enter a transaction, then no thread can proceed. The detailed rules for active( $e$ ) (or active( $T$ )) require some care. There must be exactly one inatomic expression in  $e$  (or  $T$ ), not counting possibly nested transactions inside it, and that one outermost transaction must occur in a thread’s “active position.” For example, we may be able to show active(seq(inatomic( $\circ$ , 17,  $\cdot$ ,  $\cdot$ ),  $e$ )), but we cannot show active(seq( $e$ , inatomic( $\circ$ , 17,  $\cdot$ ,  $\cdot$ ))). To summarize, proving progress requires “tight control” over the connection between each  $a$  in the program state and the state of the threads the  $a$  describes, and this control is specified with the correct( $a, T$ ) invariant. Proving preservation requires establishing this invariant after each step, particularly when a thread enters or exits a transaction.

With the ability to type-check heaps, thread-pools, and run-time expressions, we can state and prove the following two lemmas:

**Lemma 3.2 Progress** *If  $\vdash a; H; T$ , then either  $T$  is all values or  $\exists a'; H', T'$  such that  $a; H; T \rightarrow a'; H'; T'$ .*

**Lemma 3.3 Preservation** *If  $\Gamma \vdash H : \Gamma$ ,  $\Gamma; \text{tl} \vdash T$ , correct( $a, T$ ), and  $a; H; T \rightarrow a'; H'; T'$ , then there exists some  $\Gamma'$  extending  $\Gamma$  such that  $\Gamma' \vdash H' : \Gamma'$ ,  $\Gamma'; \text{tl} \vdash T'$ , and correct( $a', T'$ ).*

Since  $;\text{tl} \vdash e : \tau$  implies the initial program state type-checks (i.e.,  $\vdash \circ; ; e$ ), Theorem 3.1 is a corollary to Lemmas 3.2 and 3.3.

## 4. Weak Atomicity

In this section, we revisit the choice in our semantics that if one thread is executing a transaction, then no other thread may access the heap. Allowing such accesses is often called weak atomicity [3], meaning that a data-race between transactional and nontransactional code is allowed to violate the isolation of the transaction. It is common in STM implementations because it can simplify an implementation and/or improve its performance. Intuitively, if no data races can exist between transactional and non-transactional code, then allowing heap accesses concurrently with transactions does not lead to any additional behavior. The theorem we state in this section validates this intuition. Given the subtleties of race conditions and isolation, it is wise not to rely on intuition alone.

### 4.1 Operational Semantics

Beginning with the language in Section 2, changing the semantics to define (one notion of) weak atomicity is as simple as replacing the rules for reading and writing heap locations:

$$\frac{\text{SET}}{a; H; l := v \rightarrow a; H, l \mapsto v; v; \cdot}$$

$$\frac{\text{GET}}{a; H; !l \rightarrow a; H; H(l); \cdot}$$

$$\begin{aligned}
\tau &::= \text{int} \mid \text{ref}_t \tau \mid \tau \xrightarrow{\varepsilon} \tau' \\
t &::= \text{tl} \mid \text{ip} \\
\varepsilon &::= t \mid \emptyset \\
\Gamma &::= \cdot \mid \Gamma, x : \tau \mid \Gamma, l : (\tau, t)
\end{aligned}$$

**Figure 8.** Weak Atomicity Syntax (extends Figure 1)

That is,  $\circ$  is no longer required for heap accesses (but it is still required to enter a transaction).

This modified “weak” language clearly allows every sequence of steps the previous “strong” language does (rules GET and SET apply strictly more often), and it allows more. For example, from the program state:

$$\begin{aligned}
\circ; \quad & l_1 \mapsto 5, l_2 \mapsto 6; \\
& (\text{atomic}(\text{seq}(l_2 := 7, l_1 := !l_2))) \\
& \parallel (l_2 := 4)
\end{aligned}$$

the weak language allows a sequence of steps where the final value in  $l_1$  is 4. Therefore, the two languages are not equivalent, but there are still many programs for which they are (i.e., any result possible in one language is possible in the other). In particular, it is intuitive that for a program to distinguish the two semantics it must have thread-shared mutable data that is accessed inside and outside transactions. We now define a type system that allows only programs for which the two languages are equivalent.

## 4.2 Effect System for Ensuring Serializability

We use a type system to enforce the partition in which each memory location can be accessed outside transactions or inside transactions but not both. The syntax for types is in Figure 8. For source programs, the only difference is that reference types now carry an annotation indicating a side of a partition. For example,  $\text{ref}_{\text{ip}}(\text{ref}_{\text{tl}}\text{int})$  can be the type of an expression that produces a label that can be accessed (read or written) inside transactions and that contains a label that can be accessed outside transactions (and the pointed-to label contains an integer). Notice pointers from one side of the partition to the other are allowed. Continuing our example, if  $x$  has type  $\text{ref}_{\text{ip}}(\text{ref}_{\text{tl}}\text{int})$ , then  $(\text{atomic}(!x)) := 42$  would type-check.

Our typing judgment has the same form as before,  $\Gamma; \varepsilon \vdash e : \tau$ , meaning  $e$  has type  $\tau$  and effect  $\varepsilon$  where  $\varepsilon$  being  $\text{ip}$ ,  $\text{tl}$ , or  $\emptyset$  means  $e$  is safe inside transactions, outside transactions, or anywhere, respectively. In fact, except for disallowing  $\text{spawn}_{\text{oc}} e$  and  $\text{spawn}_{\text{ip}} e$ , most of the typing rules are identical to those in our previous effect system. The differences are in Figure 9. Rules T-SET and T-GET require the annotation on the reference type to be the same as the overall effect, which is what enforces the partition on all accesses. Notice rule T-REF does not require this equality; it is safe to allocate an outside-transactions reference inside a transaction and vice-versa. (At allocation-time the new memory is thread-local.) When extending the type system to run-time

states, the rule for labels uses  $\Gamma$  to determine the  $t$  for the accessed label, but this  $t$  need not be the same as the effect of the expression since  $t$  controls access to the label’s *contents*. As in the previous section, this extended type system is only for proof purposes; the partition and annotations are entirely conceptual (i.e., types are erasable).

The proofs of preservation and progress for this language are actually simpler than the proofs for internal parallelism. Type safety is necessary for the equivalence result we discuss next. That result is the primary reason we defined the partition-enforcing effect system.

## 4.3 Weak/Strong Equivalence Under Partition

Our primary result for this language is that any program that type-checks has the same possible behaviors under the semantics in Section 2 and the semantics in this section. Formally, letting  $\rightarrow_s^*$  mean 0 or more steps under the strong semantics and  $\rightarrow_w^*$  mean 0 or more steps under the weak semantics we have:

**Theorem 4.1 Equivalence** *If  $\cdot; \text{tl} \vdash e : \tau$ , then  $\circ; \cdot; e \rightarrow_s^* a; H; T$  if and only if  $\circ; \cdot; e \rightarrow_w^* a; H; T$ .*

In fact, the equivalence is stronger; the two semantics can produce the same states using the same number of steps. One direction of the proof is trivial because any sequence of transitions under the strong semantics is also a sequence of transitions under the weak semantics. The other direction (given a weak transition sequence, produce a strong transition sequence) is much more interesting. Space constraints require only a high-level description but the full proof is available [14].

We can strengthen the induction hypothesis as follows: If the weak semantics can produce  $a; H; T$  after  $n$  steps, then the strong semantics can produce  $a; H; T$  in  $n$  steps. Moreover, if  $a = \bullet$ , then the strong semantics can produce  $a; H; T$  in  $n$  steps using a sequence where a suffix of the sequence is the active thread entering the transaction and then taking some number of steps *without steps from any other threads interleaved*. In other words, the current transaction could have run serially at the end of the sequence.

To maintain this stronger invariant the interesting case is when the next step under the weak semantics is done by a thread not in the transaction. A key lemma lets us permute this non-transactional step to the position in the strong-semantics sequence just before the current transaction began, and the ability to permute like this without affecting the resulting program state depends precisely on the lack of memory conflicts that our type system enforces.

It is clear that this equivalence *proof* relies on notions similar to classic ideas in concurrent computation such as serializability and reducibility. Note, however, that the *theorem* is purely in terms of two operational semantics. It says that given the type system enforcing a partition, the language defined in Section 2 may be correctly implemented by the

$\Gamma; \varepsilon \vdash e : \tau$ 

$$\frac{\text{T-SET} \quad \Gamma; t \vdash e_1 : \text{ref}_t \tau \quad \Gamma; t \vdash e_2 : \tau}{\Gamma; t \vdash e_1 := e_2 : \tau}$$

$$\frac{\text{T-GET} \quad \Gamma; t \vdash e : \text{ref}_t \tau}{\Gamma; t \vdash !e : \tau}$$

$$\frac{\text{T-REF} \quad \Gamma; \varepsilon \vdash e : \tau}{\Gamma; \varepsilon \vdash \text{ref } e : \text{ref}_t \tau}$$

$$\frac{\text{T-LABEL} \quad \Gamma(l) = (\tau, t)}{\Gamma; \varepsilon \vdash l : \text{ref}_t \tau}$$

**Figure 9.** Weak Atomicity Typing (omitted rules are the same as in Figure 5)

language defined in Section 4. This result is directly useful to language implementors and does not require a notion of serializability.

#### 4.4 Toward Weaker Isolation and Partition

In practice, many STM implementations have weaker transactional semantics than those presented here. These weaknesses arise from explicit rollbacks or commits not present in our semantics. For example, if nontransactional code reads data written by a partially completed transaction that later aborts, the nontransactional thread may have read invalid data [12, 20]. Our semantics encodes an aborted transaction as one that never occurred, which means nontransactional code can never have done such a read. In future work, we intend to define a slightly more complicated semantics that allows a partially completed transaction to nondeterministically but explicitly roll back its heap changes and begin anew. We conjecture and intend to prove that the Equivalence Theorem holds for this language with no changes to the type system.

An orthogonal direction is to create a more expressive type system by relaxing the partition requirements while preserving our equivalence result. For example, thread-local or read-only data can be accessed both inside and outside transactions without invalidating equivalence. Another extension would be “partition polymorphism,” which would allow some functions to take arguments that could point into either side of the partition, depending on the call-site. This extension would require type-level variables that range over effects.

## 5. Related Work

### 5.1 Operational Semantics

The work most closely related to ours uses operational semantics to define various aspects of transactions. All work we are aware of has significantly different foci and techniques.

First, Jagannathan et al [11, 22] use a variant of Featherweight Java [10] to define a framework in which different transactional implementations (such as versioning or two-phase locking) can be embedded and proven correct by establishing a serializability result. They support internally parallel transactions by requiring each thread in the transaction to execute a commit statement before the transaction is complete. This is most similar but not identical to our

`spawnip`; they have no analogue of our other spawn flavors nor any notion of an effect system. Formally, they assume all code executes within a transaction; there is no notion of weak atomicity. Their run-time state and semantics is, in our view, more complicated, with thread identifiers, nested heaps, and traces of actions. While some of this machinery may be necessary for proving lower-level implementation strategies correct, it is less desirable for a high-level model. Although their system and ours have many technical differences, the fundamental idea of permuting independent actions arises (unsurprisingly) in both settings.

Second, Harris et al [8] present an operational semantics for STM Haskell. Like our work, it is high-level, with no explicit notion of conflicts or commits. Unlike our work, the semantics is layered such that an entire transaction occurs as one step at the outer layer, essentially using a large-step model for transactions that does not lend itself to investigating internal parallelism nor weak atomicity. Indeed, they do not have internal parallelism and the partition between mutable data accessed inside and outside transactions (enforced by the monadic typing) allows them to define strong atomicity yet implement weak atomicity. It is not particularly significant that we enforced a partition with an effect system rather than monads since there are well-known connections between the two technologies [23]. Rather, our contribution is proving that given a partition, strong and weak isolation are indistinguishable.

Third, Wojciechowski [24] proves isolation for a formal language where transactions with internal parallelism (called tasks in the work) explicitly acquire locks before accessing data and the beginning of the task must declare all the locks it might acquire. Explicit locking and version counters leads to a lower-level model and an effect system that is an extension of lock types [4]. The main theorem is essentially proving a particular low-level rollback-free transaction mechanism correct.

Finally, Liblit [13] gives an operational semantics for the hardware-based LogTM [15]. This assembly language is at a much lower level. It has neither internal parallelism nor weak atomicity.

### 5.2 Unformalized Languages

Many recent proposals for transactions in programming languages either do not discuss the effect of spawning inside a transaction or make it a dynamic error. In other words, to the extent it is considered, the most common flavor appears

to be  $\text{spawn}_{\text{tl}}$ . When designing the AtomCaml system [18], we felt that  $\text{spawn}_{\text{oc}}$  would feel natural to users. As a result,  $\text{spawn}$  on commit was selected as a reasonable method for creating new threads while executing a transaction. The Venari system for ML [7] had something close to  $\text{spawn}_{\text{ip}}$ , but it was up to the programmer to acquire locks explicitly in the style pioneered by Moss [16].

Weak atomicity has primarily been considered for its surprising pitfalls, including its incomparability with strong atomicity [3] and situations in which it leads to isolation violations that corresponding lock-based code does not [12, 9, 20]. It is widely believed that all examples of the latter require violating the partition property we considered in Section 4, which is why we plan to prove this result for various definitions of weak atomicity.

### 5.3 Other Semantics

Operational semantics gives meaning directly to source programs, which lets us consider how transactions interact with other language features, consider how a type system restricts program behavior, and provide a direct model to programmers. Other computational models, based on various notions of memory accesses or computation dependencies can prove useful for investigating properties of transactions. Recent examples include Scott’s work on specifying fairness and conflicts [19], Agrawal et al’s work on using Frigo and Luchango’s computation-centric models [5] to give semantics to open nesting [1], and Moss and Hosking’s definition of open nesting in terms of transactions’ read and write sets [17].

## 6. Conclusions and Future Work

We have presented a high-level small-step operational semantics for a programming language with transactions. We believe this language is (1) a natural way to define transactions in a manner that is precise yet not wed to implementation techniques, and (2) a starting point for defining variations that explain features and design choices. We presented three flavors of thread-spawn and proved sound a type-and-effect system that prevents spawn expressions from occurring where they cannot be successful. We also presented a weak-isolation semantics and proved it equivalent to the strong-isolation semantics assuming a particular memory partition. This result confirms that if a programming-language design can enforce such a partition, then an implementation for that language can use a weak-isolation transactional-memory system even if the language requires strong isolation.

In the future, we hope to use our approach to do the following:

- Prove our memory partition suffices for strong isolation even if aborted transactions update and then rollback heap locations.

- Incorporate open-nesting into our model and define sufficient conditions under which open-nesting is “safe” in the sense that other threads cannot determine that a transaction aborted.
- Combine our extensions for internal parallelism and weak atomicity. The operational semantics is trivial, but it is unclear if we can define a notion of memory partition that makes sense with nested internally parallel transactions.
- Define a semantics with a weaker memory model. Implicit in our current semantics is sequential consistency, which makes our semantics ill-suited to investigating troubling questions about relaxed memory models and transactions [6].

## References

- [1] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. Memory models for open-nested transactions. In *ACM SIGPLAN Workshop on Memory Systems Performance & Correctness*, 2006.
- [2] Eric Allen, David Chase, Joe Hallet, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress language specification, version 1.0 $\beta$ , 2007. <http://research.sun.com/projects/plrg/Publications/fortress1.0beta.pdf>.
- [3] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), 2006.
- [4] Cormac Flanagan and Martín Abadi. Types for safe locking. In *European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, 1999.
- [5] Matteo Frigo and Victor Luchangco. Computation-centric memory models. In *10th ACM Symposium on Parallel Algorithms and Architectures*, 1998.
- [6] Dan Grossman, Jeremy Manson, and William Pugh. What do high-level memory models mean for transactions? In *ACM SIGPLAN Workshop on Memory Systems Performance & Correctness*, 2006.
- [7] Nicholas Haines, Darrell Kindred, J. Gregory Morrisett, Scott M. Nettles, and Jeannette M. Wing. Composing first-class transactions. *ACM Transactions on Programming Languages and Systems*, 16(6):1719–1736, 1994.
- [8] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *ACM Symposium on Principles and Practice of Parallel Programming*, 2005.
- [9] Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg. McRT-Malloc: A scalable transactional memory allocator. In *International Symposium on Memory Management*, 2006.
- [10] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3), May 2001.

- [11] Suresh Jagannathan, Jan Vitek, Adam Welc, and Antony L. Hosking. A transactional object calculus. *Science of Computer Programming*, August 2005.
- [12] Jim Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2006.
- [13] Ben Liblit. An operational semantics for LogTM. Technical Report 1571, University of Wisconsin–Madison, 2006.
- [14] Katherine F. Moore and Dan Grossman. High-level small-step operational semantics for transactions (technical companion). Technical report, Univ. of Wash. Dept. of Computer Science & Engineering, 2007. Available at [http://www.cs.washington.edu/homes/kfm/atomic\\_proofs.pdf](http://www.cs.washington.edu/homes/kfm/atomic_proofs.pdf).
- [15] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in LogTM. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.
- [16] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. The MIT Press, 1985.
- [17] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: Model and preliminary architecture sketches. In *Synchronization and Concurrency in Object-Oriented Languages (SCOOOL)*, 2005.
- [18] Michael F. Ringenburt and Dan Grossman. AtomCaml: First-class atomicity via rollback. In *10th ACM International Conference on Functional Programming*, 2005.
- [19] Michael L. Scott. Sequential specification of transactional memory semantics. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006.
- [20] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steve Balensiefer, Dan Grossman, Richard Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in STM. In *ACM Conference on Programming Language Design and Implementation*, 2007.
- [21] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. Technical Report 915, Computer Science Department, University of Rochester, 2007.
- [22] Jan Vitek, Suresh Jagannathan, Adam Welc, and Antony L. Hosking. A semantic framework for designer transactions. In *European Symposium on Programming*, volume 2986 of *Lecture Notes in Computer Science*, 2004.
- [23] Philip Wadler. The marriage of effects and monads. In *3rd ACM International Conference on Functional Programming*, 1999.
- [24] Pawel T. Wojciechowski. Isolation-only transactions by typing and versioning. In *ACM International Conference on Principles and Practice of Declarative Programming*, 2005.
- [25] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.