

High-Level Small-Step Operational Semantics for Transactions

Katherine F. Moore Dan Grossman

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

Abstract

Software transactions have received significant attention as a way to simplify shared-memory concurrent programming, but insufficient focus has been given to the precise meaning of software transactions or their interaction with other language features. This work begins to rectify that situation by presenting a family of formal languages that model a wide variety of behaviors for software transactions. These languages abstract away implementation details of transactional memory, providing high-level definitions suitable for programming languages. We use small-step semantics in order to represent explicitly the interleaved execution of threads that is necessary to investigate pertinent issues.

We demonstrate the value of our core approach to modeling transactions by investigating two issues in depth. First, we consider parallel nesting, in which parallelism and transactions can nest arbitrarily. Second, we present multiple models for weak isolation, in which nontransactional code can violate the isolation of a transaction. For both, type-and-effect systems prove useful for soundly and statically restricting what computation can occur inside or outside a transaction. We prove some key language-equivalence theorems to confirm that under sufficient static restrictions, in particular that each mutable memory location is used outside transactions or inside transactions (but not both), no program can determine whether the language implementation uses weak isolation or strong isolation.

1. Introduction

1.1 The Need For Semantics

Widespread availability of multicore architectures has incited urgent interest in programming-language features that make it easier to write correct and efficient parallel programs. Software transactions are particularly appealing for shared-memory programming because they let programmers declare that an entire computation should happen “all at once” with respect to other parallel computations. A construct like `atomic (e)` means `e` should evaluate as a transaction, which provides a mutual-exclusion mechanism that avoids many of the pitfalls of locks and condition variables. It delegates to the language implementation the task of preserving the “all at once” illusion while striving to preserve parallelism among non-conflicting computations. Much recent work has investigated efficient implementation techniques [13, 15, 2, 6, 27, 21, 7, 25, 16, 32].

While this informal understanding of software transactions provides a simple high-level semantics for programmers, unfortunately, typical implementation approaches introduce complications that force programmers to abandon this simple view. As a small example, many implementations make it a dynamic error to spawn a thread while executing a transaction. If so, a library that encapsulates a parallel algorithm cannot be called in the dynamic scope of an atomic block.

Even more troubling are questions that arise from trying to give a weak-isolation semantics to transactions. Under weak isolation, nontransactional memory accesses bypass the mechanisms used to implement transactions. In recent work, we [32, 10] and others [5, 22, 17, 33] have described many surprising behaviors that can result and that cannot be explained without understanding how transactions are implemented.

As just one example, consider this code in a Java-like language:

```
Initially x=0, y=0, and b=true
Thread 1 | Thread 2 | Thread 3
atomic { | atomic { | m=x;
  if(b)  | b=false; | n=y;
    x=1; | }
  else   |
    y=1; |
}
Can m==1 and n==1?
```

Despite races between Threads 1 and 3, intuitively Thread 1 does only one assignment so `m` or `n` or both stay 0. However, weak-isolation implementations using “eager-update” [15, 2] can violate this intuition: transactions may abort-and-retry multiple times and nontransactional code may see partial results of aborted computations. Such behavior is impossible in lock-based code.

In general, the semantic ambiguities caused by weak isolation or the interaction between transactions and other language features (such as thread-creation) raise two sets of questions:

- Can we design languages that prevent undesirable behaviors while still permitting typical implementations? What restrictions must we put on source programs? How can we prove these restrictions suffice?
- If we make some errors the programmers’ responsibility, what guarantees must the language still provide? Can an illegal thread-creation or data-race lead to an arbitrary program state (like C array-bounds violations do)?

Precisely answering these questions requires rigorous semantics and proofs. Such semantics must be high-level enough to provide a simple definition to programmers yet detailed enough to incorporate relevant features. Restrictions on programs must be defined in well-understood terms, such as with a type system. Variations in semantic definitions should be compared by showing they are unobservable (via an equivalence proof) or observable (via an ex-

[Copyright notice will appear here once ‘preprint’ option is removed.]

ample program that distinguishes them). Proofs should reveal the key invariants that motivate the semantics and type systems.

1.2 Our Family of High-Level Small-Step Semantics

To meet this need, we use operational semantics to define several core languages based on a call-by-value λ -calculus with a mutable heap, threads, and transactions. Collectively, we call these languages the *AtomsFamily*. The languages differ where needed to investigate a language feature or design decision. They are not designed for reasoning about transactional-memory implementation details. Rather, they provide high-level definitions of transactions where only one transaction runs at a time. This high level matches how we want programmers to reason about transactions so it is appropriate for language definitions. At the same time, we use a small-step semantics in which transactions take multiple steps to complete. A potentially simpler transactions-in-one-step approach would make it too awkward to investigate parallel nesting or weak isolation because both features need threads to interleave while a transaction executes. As Section 7 discusses, a high-level small-step semantics distinguishes our approach from prior work.

We consider four languages in depth:

- **StrongBasic** (Section 2) is the simplest language. While a transaction executes, it cannot spawn threads and other threads that already exist may not read or write mutable heap locations.
- **StrongNestedParallel** (Section 3) extends **StrongBasic** with multiple ways to spawn threads. The different ways behave differently inside and outside transactions.
- **Weak** (Section 4) is like **StrongBasic** except nontransactional code can access the heap concurrently with a transaction, thus allowing one definition of weak isolation.
- **WeakUndo** (Section 5) is like **Weak** except a transaction may abort-and-retry by undoing its heap updates and restarting.

We also sketch two other *AtomsFamily* members: **WeakOnCommit** and **StrongUndo**. **WeakOnCommit** models transactions that can abort-and-retry, but unlike in **WeakUndo**, transactions do not update the heap until they commit. In **WeakUndo**, committing takes only one step; in **WeakOnCommit**, aborting takes only one step. In-depth investigation of **WeakOnCommit** remains future work. **StrongUndo** has the strong isolation of **StrongBasic** and the abort-and-retry of **WeakUndo**. This unusual combination is a crucial intermediate language for proving that, under a type system we define, **WeakUndo** and **StrongBasic** are suitably equivalent.

Our type systems are all very similar type-and-effect systems that classify code based on where it can run safely: only inside transactions, only outside transactions, or anywhere. For **StrongNestedParallel**, our type system ensures forms of thread-creation that make sense only inside transactions do not occur outside transactions and vice-versa. For **Weak** and **WeakUndo**, our type system enforces that the same heap location is never accessed inside and outside a transaction. A simple variant of the type system can also prevent one atomic block from executing in the dynamic scope of another.

Our languages and type systems are not exhaustive. To the contrary, we consider it a strength that new variants are easy to define and compare, nontrivial proofs notwithstanding. We expect to investigate more language features by adding to the *AtomsFamily* (see Section 6) and encourage others to do the same.

1.3 Specific Results

While generally useful, our approach has also produced several specific insights and theorems. The most important results are summarized here and explained in the remainder of the paper:

- For nested transactions to interact properly with either parallelism within transactions (**StrongNestedParallel**) or abort-and-retry (**WeakUndo**), the state of a transaction should include whether another transaction is currently executing inside it.
- A language with arbitrarily nested parallelism and transactions (**StrongNestedParallel**) can be type-safe even if certain forms for spawning threads can be used only in certain contexts.
- Weak isolation (**Weak**) and strong isolation (**StrongBasic**) are indistinguishable (i.e., the languages are equivalent) under a type system that prohibits the same heap location from being accessed inside and outside transactions. The key to the proof is showing that any computation interleaved with the current transaction would have produced the same result had it preceded the transaction.
- Weak isolation with abort-and-retry (**WeakUndo**) and strong isolation (**StrongBasic**) are indistinguishable under a similar type system as the previous result, but with some interesting caveats: (1) **WeakUndo** has some intermediate states unreachable from **StrongBasic**, (2) **WeakUndo** may allocate more memory, and (3) for simplicity we strengthen the type system to prohibit nested transactions. The key to the proof is to separate the necessary argument that the operational semantics models abort correctly.

Fortunately, the equivalence results for **Weak** and **WeakUndo** confirm conventional wisdom. Given the until recently unforeseen behaviors resulting from races between transactional and nontransactional code, it is reassuring to prove that such races are necessary for weak isolation to exhibit such behaviors. Moreover, the structure of our proofs can serve as a guide for extending the results to more sophisticated (and less obviously correct) static invariants.

Full definitions and proofs appear in our technical report [26].

2. The StrongBasic Language

This section presents the syntax and small-step operational semantics for a λ -calculus with threads, shared memory, and transactions. The language is largely a starting point for the additions and type systems in the subsequent two sections. 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 **StrongBasic**, it is essential for considering the additional thread interleavings that parallelism within transactions and weak isolation introduce.
- The semantics is *strong*. Nontransactional code cannot observe or influence partially completed transactions. We prove later that strong-isolation semantics is equivalent to weak-isolation semantics under certain conditions. One cannot do such a proof without defining both semantics.

This language does not have an explicit abort/retry. Adding this construct is easy; as in prior work [14] one simply has no evaluation rule for it. A transaction that explicitly aborts is one that can never be chosen by the nondeterministic semantics. However, this type of abort complicates stating type-safety because we would have to accommodate an abort preventing progress.

$$\begin{array}{l}
e ::= c \mid x \mid \lambda x.e \mid e_1 e_2 \mid \text{seq}(e_1, e_2) \mid \text{if } e_1 e_2 e_3 \\
\quad \mid \text{ref } e \mid e_1 := e_2 \mid !e \mid \text{spawn}_{\text{tl}} e \mid \text{atomic } e \\
\quad \mid l \mid \text{inatomic}(e) \\
v ::= c \mid \lambda x.e \mid l \\
H ::= \cdot \mid H, l \mapsto v \\
T ::= \cdot \mid T \parallel e \\
a ::= \circ \mid \bullet
\end{array}$$

Figure 1. StrongBasic Syntax

2.1 Syntax

Figure 1 presents the formal abstract syntax for our first transactional language. Most expression forms are typical for a λ -calculus with mutable references, including constants (c), variables (x), functions ($\lambda x.e$), function applications ($e_1 e_2$), sequential composition ($\text{seq}(e_1, e_2)$), conditionals ($\text{if } e_1 e_2 e_3$), memory allocation ($\text{ref } e$), assignment ($e_1 := e_2$), and dereference ($!e$). Many omitted constructs, such as records, would be straightforward additions. We also have thread-creation ($\text{spawn}_{\text{tl}} e$), where the tl indicates it must be used at top-level (not within a transaction), and atomic-blocks ($\text{atomic } e$) for executing e transactionally.

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 l , also known as addresses, to values), and T is a collection of threads. Each thread is an expression representing that thread’s remaining computation. 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, $\text{inatomic}(e)$ and l . The former represents a partially completed transaction with remaining computation e . The latter represents a heap location.

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. We do *not* 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.

2.2 Operational Semantics

Our small-step operational semantics (Figure 2) rewrites one program state $a; H; T$ to another $a'; H'; T'$. Source program e starts with $\circ; ; e$ 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.¹ 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 . For example, CONTEXT lets us derive $a; H; \text{ref}(\text{seq}(e_1, e_2)) \rightarrow a'; H'; \text{ref}(\text{seq}(e'_1, e'_2)); T$ provided $a; H; e_1 \rightarrow a'; H'; e'_1; T$.

¹We 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.

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 (GET and SET) require $a = \circ$, meaning no other thread is executing 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 access the heap. We explain below how rule INATOMIC lets the thread executing 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 needs to access the heap (which, as discussed above, requires $a = \circ$) and execute nested transactions (which requires \circ before entry and \bullet before exit), but a is \bullet while a transaction executes. That is why the hypothesis in rule INATOMIC allows any a and a' for the evaluation of the subexpression e . 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 transaction.

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 StrongBasic, but most of the errors it would prevent are standard (e.g., using an integer as a function). The only non-standard “stuck states” so far occur when a thread tries to perform a spawn inside a transaction. The type-and-effect system presented in Section 3 prevents this error.

3. The StrongNestedParallel Language

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 [28, 11, 19]. Second, if e spawns a thread (perhaps inside a library), then e and 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 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 within a transaction, we preserve correctness without letting sequential resize operations dominate performance.

In the rest of this section, we extend StrongBasic by adding several different flavors of spawn. This new language, StrongNestedParallel, demonstrates that spawn expressions within transactions can have reasonable semantics. We also present a type-and-effect system to ensure the different flavors are used sensibly.

3.1 Syntax and Operational Semantics

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

$$a; H; e \rightarrow a'; H'; e'; T$$

$$E ::= [\cdot] \mid E e \mid v E \mid \text{seq}(E, e) \mid \text{if } E e_2 e_3 \mid \text{ref } E \mid E := e \mid l := E \mid !E$$

$$\begin{array}{c} \text{CONTEXT} \\ a; H; e \rightarrow a'; H'; e'; T \\ \hline a; H; E[e] \rightarrow a'; H'; E[e']; T \end{array}$$

$$\begin{array}{c} \text{APPLY} \\ \hline a; H; (\lambda x. e) v_2 \rightarrow a; H; e[v_2/x]; \cdot \\ \text{SEQ} \\ \hline a; H; \text{seq}(v, e_2) \rightarrow a; H; e_2; \cdot \\ \text{IF-Z} \\ \hline a; H; \text{if } 0 e_2 e_3 \rightarrow a; H; e_3; \cdot \\ \text{IF-NZ} \\ \hline c \neq 0 \\ a; H; \text{if } c e_2 e_3 \rightarrow a; H; e_2; \cdot \\ \text{ALLOC} \\ \hline l \notin \text{Dom}(H) \\ a; H; \text{ref } v \rightarrow a; H, l \mapsto v; l; \cdot \\ \text{SET} \\ \hline o; H; l := v \rightarrow o; H, l \mapsto v; v; \cdot \\ \text{GET} \\ \hline o; H; !l \rightarrow o; H; H(l); \cdot \\ \text{SPAWN TL} \\ \hline a; H; \text{spawn}_{\text{tl}} e \rightarrow a; H; 0; e \\ \text{ENTER ATOMIC} \\ \hline o; H; \text{atomic } e \rightarrow \bullet; H; \text{inatomic}(e); \cdot \\ \text{EXIT ATOMIC} \\ \hline \bullet; H; \text{inatomic}(v) \rightarrow o; H; v; \cdot \\ \text{INATOMIC} \\ \hline a; H; e \rightarrow a'; H'; e'; \cdot \\ \bullet; H; \text{inatomic}(e) \rightarrow \bullet; H'; \text{inatomic}(e'); \cdot \end{array}$$

$$a; H; T \rightarrow a'; H'; T'$$

$$\begin{array}{c} \text{PROGRAM} \\ a; H; e \rightarrow a'; H'; e'; T' \\ \hline a; H; T_A \parallel e \parallel T_B \rightarrow a'; H'; T_A \parallel e' \parallel T_B \parallel T' \end{array}$$

Figure 2. StrongBasic Operational Semantics

$$\begin{array}{l} e ::= \dots \mid \text{spawn}_{\text{oc}} e \mid \text{spawn}_{\text{ip}} e \mid \text{inatomic}(a, e, T_{\text{oc}}, T_{\text{ip}}) \\ \tau ::= \text{int} \mid \text{ref } \tau \mid \tau \xrightarrow{e} \tau' \\ \varepsilon ::= \text{emp} \mid \text{ot} \mid \text{wt} \\ \Gamma ::= \cdot \mid \Gamma, x:\tau \end{array}$$

Figure 3. StrongNestedParallel Syntax (extends Figure 1)

The syntax additions are two new flavors of spawn expressions, spawn_{oc} (for “on commit”) and spawn_{ip} (for “internally parallel”). The former is allowed 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 plus spawn_{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 unadorned spawn being a synonym for one of them. For example, the current Fortress specification [4] treats spawn as spawn_{tl} , but it also has constructs for fork-join style parallelism that our model could encode with spawn_{ip} .

The inatomic expression, which 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_{oc} and T_{ip} . The a indicates whether e or any thread in T_{ip} is currently executing a transaction. T_{oc} holds 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_{tl} , T_{oc} , and T_{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, 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_{ip} , whereas T_{tl} and T_{oc} are added to the global pool of threads, i.e., spawned immediately. Therefore, it is an error to use spawn_{ip} outside a transaction.

As in StrongBasic, 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 a thread in T_{ip} can take a step, using INATOMIC DISTINGUISHED or INATOMIC PARALLEL, respectively. The only reason to distinguish e is so inatomic produces a value; in languages where the body is a statement that produces no result we could combine these two rules by including e in T_{ip} . In both rules, we evaluate some thread using a and produce an a' , H' , e' , T'_{oc} , and T'_{ip} . As in StrongBasic, evaluation inside a transaction may not spawn a top-level thread. The a' , T'_{oc} , and T'_{ip} are added to the resulting expression, and become part of the transaction’s new state. In particular, parallel threads in the transaction may produce other parallel or on-commit threads. Heap changes are propagated outward immediately, which is no problem because the outer state is \bullet .

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

Unlike in StrongBasic, nested transactions are important; they let a thread in a transaction perform multiple heap accesses atomically with respect to other threads in the transaction. Each nested transaction has an explicit a to ensure at most one of the threads is in a transaction. Because we have strong isolation, if a thread is in a transaction, then no parallel threads access the heap. However, in the innermost transaction, parallel threads may access the heap simultaneously. 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 particular type-and-effect system relies on this choice.

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

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

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

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

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

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

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

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

Figure 4. StrongNestedParallel Operational Semantics (selected rules omitted)

3.2 Type System

StrongNestedParallel has several error states. These include common type errors (e.g., treating an integer as a 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). To prove type safety, Section 3.3 extends this type system to run-time states, including labels and inatomic expressions.

The typing judgment $\Gamma; \varepsilon \vdash e : \tau$ means that (1) e has type τ where Γ provides the types of free variables, and (2) executing e only spawns threads of the flavors that the effect ε allows. A source program e type-checks if $\cdot; \text{ot} \vdash e : \tau$ for some τ . Because (1) is standard, we focus on (2), which is what makes our judgment an effect system.

The “empty” effect emp describes computations that are safe *anywhere* (i.e., inside or outside transactions); such computations spawn neither top-level nor internally-parallel threads. Spawning on-commit threads is fine because creating them never leads to dynamic errors. Effect ot describes computations safe *outside transactions*, permitting on-commit and top-level threads, and effect wt describes computations safe *within transactions*, permitting on-commit and internally-parallel threads. We do not have a “top” 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 ε . By not requiring effect emp 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 $x:\text{ref int}; \text{ot} \vdash \text{seq}(!x, \text{spawn}_{tl} 42) : \text{int}$.²

² A fine alternative is to add an effect-subsumption rule.

As expected, 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 its body’s effect is included in the effect of any call to it (see T-LAMBDA and T-APP). In T-APP, the subeffect relation allows using a function with latent effect emp in a computation with effect ot or wt . In practice, we expect most functions to type-check under emp ; 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 wt , 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 spawn_{tl} expressions must execute outside transactions, the effect of the spawn *and* of the inner expression must be ot . By contrast, all expressions *created* by spawn_{oc} are evaluated at the top level (requiring effect ot , 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 ε for spawn_{oc} . Finally, spawn_{ip} requires effect wt for the entire expression and the spawned expression because both execute only within a transaction.

If our language had expressions other than spawn_{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) $\cdot; \text{ot} \vdash e : \tau$, (2) after some number of steps $\circ; \cdot; e$ becomes $a; H; T$, and (3) not all threads*

$\varepsilon \leq \varepsilon'$	REFLEXIVE $\frac{}{\varepsilon \leq \varepsilon}$	EMPTY $\frac{}{\text{emp} \leq \varepsilon}$
$\Gamma; \varepsilon \vdash e : \tau$		
T-CONST $\frac{}{\Gamma; \varepsilon \vdash c : \text{int}}$	T-VAR $\frac{}{\Gamma; \varepsilon \vdash x : \Gamma(x)}$	T-LAMBDA $\frac{\Gamma, x; \tau_1; \varepsilon' \vdash e : \tau_2}{\Gamma; \varepsilon \vdash \lambda x. e : \tau_1 \xrightarrow{\varepsilon'} \tau_2}$
		T-APP $\frac{\Gamma; \varepsilon \vdash e_1 : \tau_1 \xrightarrow{\varepsilon'} \tau_2 \quad \Gamma; \varepsilon \vdash e_2 : \tau_1 \quad \varepsilon' \leq \varepsilon}{\Gamma; \varepsilon \vdash e_1 e_2 : \tau_2}$
T-SEQ $\frac{\Gamma; \varepsilon \vdash e_1 : \tau_1 \quad \Gamma; \varepsilon \vdash e_2 : \tau_2}{\Gamma; \varepsilon \vdash \text{seq}(e_1, e_2) : \tau_2}$	T-IF $\frac{\Gamma; \varepsilon \vdash e_1 : \text{int} \quad \Gamma; \varepsilon \vdash e_2 : \tau \quad \Gamma; \varepsilon \vdash e_3 : \tau}{\Gamma; \varepsilon \vdash \text{if } e_1 e_2 e_3 : \tau}$	T-REF $\frac{\Gamma; \varepsilon \vdash e : \tau}{\Gamma; \varepsilon \vdash \text{ref } e : \text{ref } \tau}$
		T-SET $\frac{\Gamma; \varepsilon \vdash e_1 : \text{ref } \tau \quad \Gamma; \varepsilon \vdash e_2 : \tau}{\Gamma; \varepsilon \vdash e_1 := e_2 : \tau}$
T-GET $\frac{\Gamma; \varepsilon \vdash e : \text{ref } \tau}{\Gamma; \varepsilon \vdash !e : \tau}$	T-ATOMIC $\frac{\Gamma; \text{wt} \vdash e : \tau}{\Gamma; \varepsilon \vdash \text{atomic } e : \tau}$	T-SPAWN-TL $\frac{\Gamma; \text{ot} \vdash e : \tau}{\Gamma; \text{ot} \vdash \text{spawn}_{tl} e : \text{int}}$
		T-SPAWN-OC $\frac{\Gamma; \text{ot} \vdash e : \tau}{\Gamma; \varepsilon \vdash \text{spawn}_{oc} e : \text{int}}$
		T-SPAWN-IP $\frac{\Gamma; \text{wt} \vdash e : \tau}{\Gamma; \text{wt} \vdash \text{spawn}_{ip} e : \text{int}}$

Figure 5. StrongNestedParallel Type System for Source Programs

$\Gamma; \varepsilon \vdash e : \tau$, additions and changes		
$\Gamma ::= \dots \mid \Gamma, l : \tau$	T-LABEL $\frac{\Gamma(l) = \tau}{\Gamma; \varepsilon \vdash l : \text{ref } \tau}$	T-LAMBDA $\frac{\Gamma, x; \tau_1; \varepsilon' \vdash e : \tau_2 \quad \text{not-active}(e)}{\Gamma; \varepsilon \vdash \lambda x. e : \tau_1 \xrightarrow{\varepsilon'} \tau_2}$
	T-INATOMIC $\frac{\Gamma; \text{wt} \vdash e : \tau \quad \Gamma; \text{ot} \vdash T_{oc} \quad \Gamma; \text{wt} \vdash T_{ip} \quad \text{not-active}(T_{oc}) \quad \text{correct}(a, e \parallel T_{ip})}{\Gamma; \varepsilon \vdash \text{inatomic}(a, e, T_{oc}, T_{ip}) : \tau}$	
$\Gamma \vdash H : \Gamma'$	$\Gamma; \varepsilon \vdash T$	$\vdash a; H; T$
$\frac{}{\Gamma \vdash \dots}$	$\frac{\Gamma \vdash H : \Gamma' \quad \Gamma; \varepsilon \vdash v : \tau}{\Gamma \vdash H, l \mapsto v : \Gamma', l; \tau}$	$\frac{\Gamma; \varepsilon \vdash T \quad \Gamma; \varepsilon \vdash e : \tau}{\Gamma; \varepsilon \vdash T \parallel e}$
		$\frac{\Gamma \vdash H : \Gamma \quad \Gamma; \text{ot} \vdash T \quad \text{correct}(a, T)}{\vdash a; H; T}$

Figure 6. StrongNestedParallel Type System Extensions for Program State (See also Figure 7)

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) [37]. Doing so requires extending the type system to run-time states $a; H; T$ including the expression forms not present in source programs. This extended system, presented in Figure 6, exists only for the proof. Proof details are in an available technical report [26]; here we sketch the interesting invariants that 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 ($\Gamma \vdash H : \Gamma$),³ and (2) each thread type-checks under effect ot and the labels in the heap ($\Gamma; \text{ot} \vdash T$). Note our definition of Γ now includes types for labels. Also note that 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 technical point ensures a function body never contains a partially completed trans-

action. 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 T-INATOMIC has several subtleties. Because e and T_{ip} evaluate within a transaction, they must have effect wt . Similarly, T_{oc} will be evaluated at the top level, so it must have effect ot . As with atomic, the overall effect of inatomic is unconstrained to allow nesting. As with function bodies, the not-yet-running threads T_{oc} must not contain inatomic expressions.

The final definition to explain is $\text{correct}(a, T)$, which is used to type-check program states and inatomic expressions. This judgment, defined formally in the technical report and summarized in Figure 7, is essential when showing that each a is correct — $a = \bullet$ if 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 $\text{active}(e)$ (and $\text{active}(T)$) require some care. There must

³Using Γ twice is a technical trick to allow cycles in the heap.

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

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 key 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 $\Gamma \vdash H : \Gamma$, correct(a, T), $\Gamma; \text{ot} \vdash T$, and $a; H; T \rightarrow a'; H'; T'$, then there exists some Γ' extending Γ such that $\Gamma' \vdash H' : \Gamma'$, correct(a', T'), and $\Gamma'; \text{ot} \vdash T'$.*

Because $;\text{ot} \vdash e : \tau$ 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. The Weak Language

In this section, we revisit the choice in StrongBasic that if one thread is executing a transaction, then other threads may not access the heap. Allowing such accesses is often called weak atomicity [5], meaning a data-race between transactional and nontransactional code is allowed to violate a transaction’s isolation, so we call the new language Weak. Weak isolation is common in software implementations of transactional memory because it is simpler to implement and usually improves 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 main theorem we present validates this intuition. Given the subtleties of race conditions and isolation, it is wise not to rely on intuition alone.

4.1 Operational Semantics

Changing StrongBasic to produce Weak is extremely simple; we leave the syntax unchanged and replace the operational rules for reading and writing heap locations:

GET	SET
$a; H; !l \rightarrow a; H; H(l); \cdot$	$a; H; l := v \rightarrow a; H, l \mapsto v; v; \cdot$

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

This new language clearly allows every sequence of steps StrongBasic does (rules GET and SET apply strictly more often), and it allows more. For example, from the program state:

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

τ	::=	int		ref _t τ	$\tau \xrightarrow{\varepsilon} \tau'$
t	::=	ot		wt	
ε	::=	emp		t	
Γ	::=	\cdot		$\Gamma, x:\tau$	$\Gamma, l:(\tau, t)$

Figure 8. Weak Type-System Syntax

Weak 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 Type-And-Effect System for Ensuring Serializability

Our type-and-effect system enforces a partition in which each memory location can be accessed outside transactions or inside transactions but not both. More expressive type systems are possible (see Section 6), but this system suffices for showing the key ideas in proving equivalence assuming a static discipline. It also corresponds to the partition enforced by the monads in STM Haskell [14].

The syntax for types is in Figure 8. Our effects are the same as in StrongNestedParallel; the difference is we now use them to restrict heap accesses. As such, reference types now carry an annotation indicating a side of a partition. For example, ref_{wt}(ref_{ot} int) is 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 ref_{wt}(ref_{ot} int), then (atomic(! x)) := 42 would type-check.

Our typing judgment has the same form as before, $\Gamma; \varepsilon \vdash e : \tau$, meaning e has type τ and effect ε where ε being wt, ot, or emp means e is safe inside transactions, outside transactions, or anywhere, respectively. In fact, except for disallowing spawn_{oc} e and spawn_{ip} e (because like in StrongBasic we have only top-level spawn), most of the typing rules are identical to those in StrongNestedParallel. 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 has no such requirement; it is safe to allocate an ot reference inside a transaction and vice-versa. (At allocation-time the new memory is thread-local.) To extend the type system to run-time states, T-LABEL uses Γ to determine the t for the accessed label, but this t need not be the same as the effect of the expression because t controls access to the label’s contents. As before, we extend the type system only for the proofs; the partition and annotations are entirely conceptual (i.e., types are erasable).

The proofs of preservation and progress for Weak are similar to the proofs for StrongNestedParallel, but here type safety ensures evaluation preserves the heap partition. This invariant is necessary for the equivalence result we discuss next.

4.3 Weak/Strong Equivalence Under Partition

Our primary result is that any program that type-checks has the same possible behaviors under StrongBasic and Weak. 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 $;\text{ot} \vdash e : \tau$, then $\circ; ; e \rightarrow_s^* a; H; T$ if and only if $\circ; ; e \rightarrow_w^* a; H; T$.*

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

$$\begin{array}{c} \text{T-SET} \\ \frac{\Gamma; t \vdash e_1 : \text{ref}_t \tau \quad \Gamma; t \vdash e_2 : \tau}{\Gamma; t \vdash e_1 := e_2 : \tau} \end{array} \quad \begin{array}{c} \text{T-GET} \\ \frac{\Gamma; t \vdash e : \text{ref}_t \tau}{\Gamma; t \vdash !e : \tau} \end{array} \quad \begin{array}{c} \text{T-REF} \\ \frac{\Gamma; \varepsilon \vdash e : \tau}{\Gamma; \varepsilon \vdash \text{ref } e : \text{ref}_t \tau} \end{array} \quad \begin{array}{c} \text{T-LABEL} \\ \frac{\Gamma(l) = (\tau, t)}{\Gamma; \varepsilon \vdash l : \text{ref}_t \tau} \end{array} \quad \begin{array}{c} \text{T-INATOMIC} \\ \frac{\Gamma; \text{wt} \vdash e : \tau \quad \text{correct}(a, e)}{\Gamma; \varepsilon \vdash \text{inatomic}(e) : \tau} \end{array}$$

Figure 9. Weak Type System (omitted rules and definitions are the same as in Figures 5, 6, and 7)

In fact, the equivalence is stronger; the two semantics can produce the same states in the same number of steps. One direction of the proof is trivial: any sequence of transitions under `StrongBasic` is also a sequence of transitions under `Weak`. The other direction (given a weak transition sequence, produce a strong transition sequence) is much more interesting. Space constraints allow only a high-level description but the full proof is available [26].

We strengthen the induction hypothesis as follows: If `Weak` can produce $a; H; T$ in n steps, then `StrongBasic` can produce $a; H; T$ in n steps. Moreover, if $a = \bullet$, then `StrongBasic` 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.

In maintaining this stronger invariant, the interesting case is when the next step under `Weak` 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 language defined in Section 4. This result is directly useful to language implementors and does not require a notion of serializability.

However, the `Weak` language remains unrealistic because transactions never execute partially and then abort, a significant complication the following section addresses.

5. The WeakUndo Language

In practice, many implementations of transactions employ abort-and-retry, undoing any changes a transaction has made and starting again from the original atomic expression. There are various reasons to do this, such as avoiding a memory conflict with a concurrently executing transaction or not having a transaction stay live across a thread context-switch. Because these and other reasons are low-level details not visible in the source language,⁴ the best way to model abort-and-retry at a high-level is to let a transaction undo its changes nondeterministically, i.e., at any point. Our `WeakUndo` language does precisely this, revealing interesting interactions with both nested transactions and weak isolation.

5.1 Syntax and Operational Semantics

Figure 10 presents the new syntax and Figure 11 presents the new evaluation rules.

Our source language is the same as for `StrongBasic` and `Weak`. At run-time, transactions need extra state to enable rollback, so we

⁴In the case of memory conflicts, false-sharing issues arising detecting conflicts at a coarse granularity (e.g., using hashing or cache-lines) can make a conflict unpredictable at the source level.

$$e ::= \dots \mid \text{inatomic}(a, e, H_{\log}, e_0) \mid \text{inrollback}(H_{\log}, e_0)$$

Figure 10. WeakUndo Syntax (extends Figure 1)

have `inatomic`(a, e, H_{\log}, e_0). The a indicates whether the computation e currently has a nested transaction like in `StrongNestedParallel`. This is important because we cannot perform an undo when there is a nested transaction; it must be completed or undone first. The H_{\log} is a log of labels written to and the values they had before the assignment. To undo a transaction’s memory effects, we use the log to undo the assignments in last-in-first-out order. Syntactically, H_{\log} maps labels to values like a heap, but unlike a heap it is the *first* (or *leftmost*) entry for a label that holds the relevant value. Finally, e_0 is the transaction’s initial expression, so a transaction starts by `atomic` e_0 becoming `inatomic`(\circ, e_0, \cdot, e_0) (i.e., no nested transaction and an empty log).

To undo a transaction, we roll back the heap assignments one at a time using the log. The syntax `inrollback`(H_{\log}, e_0) maintains the state of an undo. It is like `inatomic` except we do not need any a or e . The log gets smaller as evaluation rolls back entries.

For the operational semantics, an evaluation step for one thread produces H_{\log} , which contains the entries that must be appended to the log in the nearest enclosing transaction. `CONTEXT` propagates such entries and rules like `SEQ` (not shown) produce the empty log \cdot . `SET` produces the one-entry log $l \mapsto H(l)$, i.e., l maps to the value before the assignment while updating the heap as usual. `GET` produces an empty log; there is nothing to undo. Most importantly, `INATOMIC` appends the log produced by the subexpression evaluation⁵ and propagates the empty-log. If a transaction eventually aborts, it never propagates any log entries. Else, `COMMIT` propagates the entire log for the next enclosing transaction in one step. Because logs are unneeded outside transactions, `PROGRAM` does not use the log its hypothesis produces.

A rollback occurs by using `ENTER ROLLBACK`, then using one `DO ROLLBACK` for each log entry, and finally using `COMPLETE ROLLBACK`. To begin rollback, there must be no nested transaction else the entries in its log would be lost. The \circ in the `inatomic` expression in `ENTER ROLLBACK` enforces this requirement. During rollback, the top-level transaction state remains \bullet ; we cannot start another transaction until the rollback is complete. Finally, `COMPLETE ROLLBACK` produces an atomic expression ready to (re)execute, but another transaction may run first.

After rollback all labels have the values they had before the transaction began, but the heap is not exactly the same. Memory allocation (rule `ALLOC`, not shown in Figure 11) does *not* produce a log entry and rollback does *not* remove the new label from the heap. However, it will be unreachable (i.e., garbage) after the transaction rolls back.

Figure 12 shows an example trace possible in `WeakUndo` but not in `Weak` or `StrongBasic`; it is similar to the example in Section 1. Each row represents a program state (not every state is

⁵By $H_{\log} H'_{\log}$ we mean the log where H'_{\log} follows H_{\log} , so the last-in-first-out undo will process the H'_{\log} entries before the H_{\log} entries.

$$a; H; e \rightarrow a'; H'; e'; T; H_{\log}$$

$\frac{\text{CONTEXT} \quad a; H; e \rightarrow a'; H'; e'; T; H_{\log}}{a; H; E[e] \rightarrow a'; H'; E[e']; T; H_{\log}}$	$\frac{\text{SET} \quad a; H; l := v \rightarrow a; H, l \mapsto v; v; \cdot; l \mapsto H(l)}{a; H; l := v \rightarrow a; H, l \mapsto v; v; \cdot; l \mapsto H(l)}$	$\frac{\text{GET} \quad a; H; l \rightarrow a; H; H(l); \cdot; \cdot}{a; H; l \rightarrow a; H; H(l); \cdot; \cdot}$
$\frac{\text{ENTER ATOMIC} \quad \circ; H; \text{atomic } e \rightarrow \bullet; H; \text{inatomic}(\circ, e, \cdot, e); \cdot; \cdot}{\circ; H; \text{atomic } e \rightarrow \bullet; H; \text{inatomic}(\circ, e, \cdot, e); \cdot; \cdot}$	$\frac{\text{INATOMIC} \quad a; H; e \rightarrow a'; H'; e'; \cdot; H'_{\log}}{\bullet; H; \text{inatomic}(a, e, H_{\log}, e_0) \rightarrow \bullet; H'; \text{inatomic}(a', e', H_{\log} H'_{\log}, e_0); \cdot; \cdot}$	
$\frac{\text{COMMIT} \quad \bullet; H; \text{inatomic}(\circ, v, H_{\log}, e_0) \rightarrow \circ; H; v; \cdot; H_{\log}}{\bullet; H; \text{inatomic}(\circ, v, H_{\log}, e_0) \rightarrow \circ; H; v; \cdot; H_{\log}}$	$\frac{\text{ENTER ROLLBACK} \quad \bullet; H; \text{inatomic}(\circ, e, H_{\log}, e_0) \rightarrow \bullet; H; \text{inrollback}(H_{\log}, e_0); \cdot; \cdot}{\bullet; H; \text{inatomic}(\circ, e, H_{\log}, e_0) \rightarrow \bullet; H; \text{inrollback}(H_{\log}, e_0); \cdot; \cdot}$	
$\frac{\text{DO ROLLBACK} \quad \bullet; H; \text{inrollback}(H_{\log}, l \mapsto v_{\text{old}}, e_0) \rightarrow \bullet; H, l \mapsto v_{\text{old}}; \text{inrollback}(H_{\log}, e_0); \cdot; \cdot}{\bullet; H; \text{inrollback}(H_{\log}, l \mapsto v_{\text{old}}, e_0) \rightarrow \bullet; H, l \mapsto v_{\text{old}}; \text{inrollback}(H_{\log}, e_0); \cdot; \cdot}$	$\frac{\text{COMPLETE ROLLBACK} \quad \bullet; H; \text{inrollback}(\cdot, e_0) \rightarrow \circ; H; \text{atomic } e_0; \cdot; \cdot}{\bullet; H; \text{inrollback}(\cdot, e_0) \rightarrow \circ; H; \text{atomic } e_0; \cdot; \cdot}$	
$a; H; T \rightarrow a'; H'; T'$	$\frac{\text{PROGRAM} \quad a; H; e \rightarrow a'; H'; e'; T; H_{\log}}{a; H; T_A \parallel e \parallel T_B \rightarrow a'; H'; T_A \parallel e' \parallel T_B \parallel T}$	

Figure 11. WeakUndo Operational Semantics (selected rules omitted)

	a	l_x	l_y	l_z	l_m	l_n	Thread 1	Thread 2
\rightarrow^*	\circ	0	0	0	0	0	atomic (if ($!l_x$) ($l_y := 1$) ($l_z := 1$)))	$l_x := 1; l_n := !l_z; l_m := !l_y$
\rightarrow^*	\bullet	0	0	1	0	0	inatomic($\circ, 1, l_z \mapsto 0$, (if ($!l_x$) ($l_y := 1$) ($l_z := 1$))))	$l_x := 1; l_n := !l_z; l_m := !l_y$
\rightarrow^*	\bullet	1	0	1	0	1	inatomic($\circ, 1, l_z \mapsto 0$, (if ($!l_x$) ($l_y := 1$) ($l_z := 1$))))	$l_m := !l_y$
\rightarrow^*	\circ	1	0	0	0	1	atomic (if ($!l_x$) ($l_y := 1$) ($l_z := 1$)))	$l_m := !l_y$
\rightarrow^*	\circ	1	1	0	0	1	1	1
\rightarrow^*	\circ	1	1	0	1	1	1	1

Figure 12. Selected states from a WeakUndo trace, starting with $\circ; H; T$ where H maps each label to 0 and T contains Thread 1 and Thread 2. In the end, l_m and l_n map to 1. Under Weak, Thread 1 could not change both l_y and l_z .

shown). The transaction in Thread 1 rolls back between the third and fourth rows and its reexecution takes the other conditional branch. However, Thread 2 uses nontransactional code to see a write from Thread 1 before that write is rolled back. Notice the initial state shown would not type-check under our effect system for partitioning the heap because l_x, l_y , and l_z are all accessed inside (Thread 1) and outside (Thread 2) transactions.

5.2 Type-And-Effect System

For source programs, we can use the same type system we do for Weak to ensure the same memory is not used inside and outside transactions. Extending the type system to the new inatomic and inrollback forms requires maintaining a number of technical invariants. For example, all log entries must be for labels with effect wt. Given these details (see the technical report [26]) Preservation and Progress hold as in the other AtomsFamily languages.

Though we believe WeakUndo is equivalent to StrongBasic, for simplicity our current proof of this result does not consider nested transactions. Fortunately, formalizing this simplification requires only a small change. We can forbid nested transactions by type-checking atomic e only under effect ot rather than ε :

$$\frac{\text{T-ATOMIC} \quad \Gamma; \text{wt} \vdash e : \tau}{\Gamma; \text{ot} \vdash \text{atomic } e : \tau}$$

Type-checking inatomic requires an analogous change. Preservation and Progress also hold for this more restrictive type system.

5.3 WeakUndo/Strong Equivalence

Using the type system described above and letting \rightarrow_s^* mean 0 or more steps under StrongBasic and \rightarrow_{wu}^* mean 0 or more steps under WeakUndo, we have the following theorem:

Theorem 5.1 Equivalence *If $\cdot; \text{ot} \vdash e : \tau$, then:*

(1) *If $\circ; \cdot; e \rightarrow_s^* \circ; H; T$, then $\circ; \cdot; e \rightarrow_{wu}^* \circ; H; T$ and*

(2) *If $\circ; \cdot; e \rightarrow_{wu}^* \circ; H; T$, then there exists an H' such that $\circ; \cdot; e \rightarrow_s^* \circ; H'; T$ and for all l and v , if $H'(l) = v$, then $H(l) = v$.*

This theorem is weaker than the analogous theorem for Weak in two interesting ways. First, WeakUndo may produce a larger heap (the domain of H may exceed the domain of H' , but H restricted to the domain of H' must be H') because aborted transactions can generate garbage. Second, the equivalence holds only when a is \circ , i.e., no transaction is executing. If a is \bullet , then WeakUndo may be performing a rollback and StrongBasic has no corresponding state. Also, as a technical point, the different syntax for inatomic in the languages precludes having syntactically equal programs whenever a transaction is executing.

While part (1) of the theorem is trivial, part (2) is not. The proof, detailed in the technical report [26] and briefly summarized here, cannot ignore states where a is \bullet . Instead, we must show that transactions in WeakUndo are serializable (as in the proof for Weak)

and that rollback is correct (produces a state close enough to the one before the transaction began). Because the latter is much easier to show under strong isolation, we define an intermediary language StrongUndo in which we have inrollback and the corresponding evaluation rules as in WeakUndo but when one thread is executing a transaction, no other thread can execute. We then have these two lemmas (with \rightarrow_{su}^* for evaluation under StrongUndo):

Lemma 5.2 *If $\cdot; \text{ot} \vdash e : \tau$ and $\circ; \cdot; e \rightarrow_{wu}^* a; H; T$, then $\circ; \cdot e \rightarrow_{su}^* a; H; T$.*

Lemma 5.3 *If $\cdot; \text{ot} \vdash e : \tau$ and $\circ; \cdot; e \rightarrow_{su}^* \circ; H; T$, then there exists an H' such that $\circ; \cdot; e \rightarrow_s^* \circ; H'; T$ and for all l and v , if $H'(l) = v$, then $H(l) = v$.*

Proving the first lemma follows exactly the proof strategy described in Section 4.3 for Weak and StrongBasic, with additional cases for the rollback steps. Proving the second lemma requires a strengthened induction hypothesis arguing that whenever StrongUndo is executing a transaction, all the following hold:

- If StrongUndo is not rolling the transaction back, then StrongBasic could get to a similar state.
- If StrongUndo is not rolling the transaction back, but it chose to from this point, then it would produce a state just like before the transaction started (plus possible garbage).
- If StrongUndo is rolling the transaction back, then after completing the rollback it will have a state just like before the transaction started (plus possible garbage).

As a corollary, Weak and WeakUndo are equivalent for well-typed programs because both are equivalent to StrongBasic. We were surprised that we did not prove WeakUndo equivalent to Weak directly, but it is not clear to us how to do so. StrongUndo turned out to be a crucial technical tool. Abadi et al. independently reached a very similar conclusion [1], which indicates that this approach is indeed the natural one.

6. Future Work

Because the AtomsFamily approach is amenable to investigating different features, there are many directions for future work. We first describe a language that is in many ways dual to WeakUndo but for which we have not yet proven relevant theorems. We then consider other ways to define transactional semantics, make our type systems more expressive, or add new language features.

6.1 The WeakOnCommit Language

Instead of supporting abort-and-retry by keeping a log of old values, we can maintain a private copy of updated heap values in a transaction and propagate updated values only when a transaction commits. We have fully defined the syntax and operational semantics of such a language (see [26]), which we call WeakOnCommit, but have not yet formalized a full type system for this language. This section sketches WeakOnCommit and its relation to WeakUndo.

At run-time, transactions have the form $\text{inatomic}(e, H, e_0)$ where e , and e_0 are like in WeakUndo, but H is an “on-commit” heap containing labels allocated or updated by the transaction so far. Inside a transaction, any assignment or allocation is propagated out only to the innermost transaction, where it is added to the H . To read a reference, the operational semantics look first at the heap for the closest containing transaction and then in the next closest heap if it is not there.

To handle this gracefully in a formal operational semantics, the judgment for evaluating an expression uses a *stack of heaps*

S where S is defined inductively as one heap H or a stack $S::H$ where H is the shallowest stack element. We then have the judgment $a; S; e \rightarrow a'; S'; e'; T$. Outside of a transaction, S is just the outermost heap, i.e., the H in the program state $a; H; T$. Inside a transaction, we have a deeper stack:

$$\frac{\text{INATOMIC} \quad a; S::H; e \rightarrow a'; S::H'; e'; \cdot}{\bullet; S; \text{inatomic}(e, H, e_0) \rightarrow \bullet; S; \text{inatomic}(e', H', e_0); \cdot}$$

Evaluation of e can change H (which is empty when the transaction starts) but not S . For example, the rule for assignment is:

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

However, evaluation needs the entire stack $S::H$ because the rule for $!$ searches the stack in order from H outward.

Aborting a transaction in WeakOnCommit takes only one step and can apply even if there is a nested transaction:

$$\frac{\text{ROLLBACK}}{\bullet; S; \text{inatomic}(e, H, e_0) \rightarrow \circ; S; \text{atomic } e_0; \cdot}$$

On the other hand, to commit a transaction we use the new syntax form $\text{incommit}(H, v)$. A transaction $\text{inatomic}(v, H, e_0)$ can step to $\text{incommit}(H, v)$, after which abort is impossible. Then elements of H are propagated out one label at a time, removing them from H , and finally $\text{incommit}(\cdot, v)$ becomes v . Whereas in WeakUndo the heap in $\text{inrollback}(H, e_0)$ maps labels to old values, in WeakOnCommit the H in $\text{incommit}(H, v)$ maps labels to new values.

For programs that do not type-check under our type-and-effect system, strange behaviors can arise. As in actual implementations, we have defined the in-commit rules to propagate the new values for the labels in an arbitrary order.⁶ Hence, nontransactional code racing with $\text{atomic}(\text{seq}(x := 1, y := 1))$ could see the assignment to y before the assignment to x . See our prior work for how this flexibility leads to strange results [32]. In future work, we intend to prove WeakOnCommit equivalent to StrongBasic for well-typed programs and to explore the extent to which WeakUndo and WeakOnCommit are equivalent even for ill-typed programs.

6.2 More Permissive Semantics

There are several ways to relax the type-and-effect system for Weak and WeakUndo without invalidating our equivalence results. For example, we could have invariants for thread-local or read-only data because both can be accessed inside and outside transactions without interleaving with other threads causing problems. 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.

The AtomsFamily can also be extended with languages that have more permissive dynamic semantics (i.e., allow more behaviors). For example, we could support open-nesting by having a construct $\text{open}(e)$ where the effects of e are never undone even if an enclosing transaction aborts [29]. Hopefully we can define sufficient conditions under which open-nesting is “safe” in the sense that other threads cannot determine that a transaction aborted. We would also like to investigate relaxed memory models [10, 24], which can be awkward because it is unnatural for a formal operational semantics not to be sequentially consistent.

⁶Implementations have strange orders if, for example, they use hashables.

6.3 Other Language Interactions

More languages similar to the `AtomsFamily` could allow additional constructs and combinations thereof that merit investigation. For example, combining the weak isolation of `Weak` and the nested parallelism of `StrongNestedParallel` is straightforward for the semantics, but the type system adjustments needed to preserve equivalence are currently unclear. In addition, the interaction of transactions with exceptions [12, 30] or first-class continuations [20] needs to be defined precisely. Programs using transactions also need fairness guarantees from the thread scheduler and conflict manager, and integrating such guarantees into our models would be valuable.

7. Related Work

7.1 Prior Work on Operational Semantics

The prior work most closely related to ours uses operational semantics to define various aspects of transactions. All such work we are aware of has significantly different foci and techniques, focusing either implementation-level issues or modeling transactions as a single computational step.

First, Jagannathan et al. [19, 34] use a variant of Featherweight Java [18] 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 parallelism within transactions by requiring each thread in the transaction to execute a commit statement for the transaction to complete. This is similar but not identical to our `spawnip`; they have no analogue of our other `spawn` flavors nor any effect system. Formally, they assume all code executes within a transaction; there is no notion of weak isolation. 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. Though their system and ours have many technical differences, the fundamental idea of permuting independent actions arises (unsurprisingly) in both settings.

Second, Harris et al. [14] present an operational semantics for STM Haskell. Like our work, it is high-level, with only one transaction executing at a time. However, 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 nested parallelism nor weak isolation. Indeed, they do not have nested parallelism and the partition between mutable data accessed inside and outside transactions (enforced by a monad) lets them define strong isolation yet implement weak isolation. It is not significant that we enforced a partition with an effect system rather than monads as there are well-known connections between the two technologies [35]. Rather, our contribution is proving that given a partition, strong and weak isolation are indistinguishable.

Third, Wojciechowski [36] proves isolation for a formal language where transactions with nested 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 [8]. The main theorem essentially proves a particular low-level rollback-free transaction mechanism correct.

Finally, Liblit [23] gives an operational semantics for the hardware-based LogTM [27]. This assembly language is at a much lower level. It has neither nested parallelism nor weak isolation.

7.2 Concurrent Work on Operational Semantics

We recently learned that Abadi et al. [1] developed a small-step operational model for transactions at the same time as our work.

Among various differences in the basic approach, the most significant is that we have a lexically scoped transaction (`atomic (e)`) whereas they have primitives for starting and ending transactions. Because they prohibit starting a transaction within another, they do not have any notion of nested transactions. On the other hand, they use an effect system to prevent using the end-transaction primitive when no transaction is executing.

Both projects investigated weak isolation with reassuringly similar results. In our terms, Abadi et al. also proved `WeakUndo` equivalent to `StrongBasic` and even followed the approach of using `StrongUndo` as an intermediate point. In proving `Weak` equivalent to `StrongBasic`, they used a semantic notion of memory conflict rather than our more restrictive syntactic type-and-effect system.

Aside from weak isolation, the projects have considered different extensions. Abadi et al. have not considered parallelism within transactions. Instead, they have considered a model where multiple threads can execute transactions simultaneously but any conflict aborts all the transactions. (In practice, implementations abort only one transaction when they detect a conflict.)

7.3 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 is `spawntl`. When designing the `AtomCaml` system [30], we felt `spawnoc` would be most natural, but it was the only option. The Venari system for ML [11] had something close to `spawnip`, but it was up to the programmer to acquire locks explicitly in the style pioneered by Moss [28].

Weak isolation has primarily been considered for its surprising pitfalls, including its incomparability with strong isolation [5] and situations in which it leads to isolation violations that corresponding lock-based code does not [22, 17, 32]. It is believed that all examples of the latter require violating the partition property we defined in Section 4, which is why we proved this result for `Weak` and `WeakUndo`.

7.4 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 [31], Agrawal et al.'s work on using Frigo and Luchango's computation-centric models [9] to give semantics to open nesting [3], and Moss and Hosking's definition of open nesting in terms of transactions' read and write sets [29].

8. Conclusions

The `AtomsFamily` is a collection of core languages that uses small-step operational semantics to model software transactions. We have used this approach to investigate the precise meaning of both parallelism within transactions and weak isolation. Our approach reveals subtle differences among similar semantics without exposing implementation details of transactional memory. We have used type-and-effect systems to restrict programs so that evaluation does not get stuck and different `AtomsFamily` languages are equivalent.

In general, our work brings a needed level of rigor to the definition of programming languages containing software transactions. We have provided several possible definitions, an approach that makes defining additional possibilities straightforward, and metatheory proofs revealing the key invariants that make transac-

tions work as expected. This work provides the necessary foundation for evaluating the correctness of language implementations.

References

- [1] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Personal communication.
- [2] Ali-Reza Adl-Tabatabai, Brian Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *ACM Conference on Programming Language Design and Implementation*, pages 26–37, June 2006.
- [3] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. Memory models for open-nested transactions. In *ACM SIGPLAN Workshop on Memory Systems Performance & Correctness*, 2006.
- [4] 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 β , 2007. <http://research.sun.com/projects/plrg/Publications/fortress1.0beta.pdf>.
- [5] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), 2006.
- [6] Brian D. Carlstrom, JaeWoong Chung, Austen McDonald, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. The Atomos transactional programming language. In *ACM Conference on Programming Language Design and Implementation*, pages 1–13, June 2006.
- [7] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.
- [8] Cormac Flanagan and Martín Abadi. Types for safe locking. In *European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, 1999.
- [9] Matteo Frigo and Victor Luchangco. Computation-centric memory models. In *10th ACM Symposium on Parallel Algorithms and Architectures*, 1998.
- [10] 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.
- [11] 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.
- [12] Tim Harris. Exceptions and side-effects in atomic blocks. In *PODC Workshop on Concurrency and Synchronization in Java Programs*, July 2004.
- [13] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2003.
- [14] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *ACM Symposium on Principles and Practice of Parallel Programming*, 2005.
- [15] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *ACM Conference on Programming Language Design and Implementation*, pages 14–25, June 2006.
- [16] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2006.
- [17] 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.
- [18] 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.
- [19] Suresh Jagannathan, Jan Vitek, Adam Welc, and Antony L. Hosking. A transactional object calculus. *Science of Computer Programming*, August 2005.
- [20] Aaron Kimball and Dan Grossman. Software transactions meet first-class continuations, June 2007. Submitted for workshop publication.
- [21] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *ACM Symposium on Principles and Practice of Parallel Programming*, 2006.
- [22] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2006.
- [23] Ben Liblit. An operational semantics for LogTM. Technical Report 1571, University of Wisconsin–Madison, 2006.
- [24] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *32nd ACM Symposium on Principles of Programming Languages*, January 2005.
- [25] Virendra J. Marathe, William N. Scherer, and Michael L. Scott. Adaptive software transactional memory. In *International Symposium on Distributed Computing*, 2005.
- [26] 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/atomsfamily_proofs.pdf.
- [27] 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.
- [28] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. The MIT Press, 1985.
- [29] 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.
- [30] Michael F. Ringenburt and Dan Grossman. AtomCaml: First-class atomicity via rollback. In *10th ACM International Conference on Functional Programming*, 2005.
- [31] Michael L. Scott. Sequential specification of transactional memory semantics. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006.
- [32] 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.
- [33] 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.
- [34] 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.
- [35] Philip Wadler. The marriage of effects and monads. In *3rd ACM International Conference on Functional Programming*, 1999.
- [36] Pawel T. Wojciechowski. Isolation-only transactions by typing and versioning. In *ACM International Conference on Principles and Practice of Declarative Programming*, 2005.
- [37] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.