Under consideration for publication in Math. Struct. in Comp. Science

Static Typing and Dynamic Versioning for Safe Pessimistic Concurrency Control

PAWEŁ T. WOJCIECHOWSKI

Institute of Computing Science, Poznań University of Technology, Poland.
ptw@cs.put.poznan.pl

Received April 4, 2012

Contents

1. Introduction

Concurrent programming is notoriously difficult. The low-level abstractions such as fine-grained locks are intricate and make it hard to write reliable programs and to reason about program correctness. In recent years, there has been a growing interest in adopting atomic transactions to general-purpose programming languages (see e.g., (HF03; HMPH05) among others). They enable an elegant declarative style of concurrency control in which programmers use high-level programming constructs to indicate the safety properties that they require. The runtime system ensures that the concurrent transactions commit atomically and exactly once. A lot of work in this area is based on the concept of *Software Transactional Memory (STM)* (ST95), relaying on optimistic concurrency control. Optimistic concurrency control allows more parallelism than locks since transaction operations are executed without blocking. In case of any conflicts, transactions are rolled back and reexecuted. This approach is, however, problematic. Firstly, native methods having irrevocable effects cannot be used freely inside transactions. Secondly, high contention in accessing shared data by a large number of concurrent transactions will cause many conflicts. The conflicting transactions have to be aborted and reexecuted decreasing the overall system throughput. Various contention managers have been proposed (see e.g., (SS05)). They may help to decrease the transaction abort rate but they are not able to solve the problem of methods with irrevocable effects. Thus, some STM implementations combine optimistic and pessimistic concurrency control (see e.g. (NWAT$^+$08)), where the latter is used for transactions that cannot be executed optimistically. In general, using pessimistic concurrency control based on locks eliminates conflicts between transactions, so they are never forced to abort. However, standard locking principle requires some additional means to deal with deadlocks statically or dynamically.

This paper defines a calculus of atomic transactions with pessimistic concurrency control based on *versioning*, where versioning replaces locks by *versioning locks* (or *velocks*) that give transactions safe and deadlock-free exclusive access to shared resources with isolation guarantee (explained below). Deadlock can still occur due to incorrect use of locks *inside* a transaction. This problem is, however, orthogonal to our design, and existing solutions of deadlock avoidance or detection can be applied to solve it. The calculus is equipped with a structural operational semantics, describing a high-level semantics of the programming language constructs and a low-level semantics of a concurrency controller. The key concept of our design is the use of data derived from a program statically to support efficient and safe transaction execution at runtime. We present a first-order type system that can statically verify input data for an example versioning algorithm that is used to implement the concurrency controller. We have used one of the simplest algo-

rithms possible. It does not permit much concurrency between the transaction threads that may access the same data but it makes the presentation of the calculus simpler.

We have used the operational semantics of our calculus to formalize and prove type soundness and the runtime correctness of our versioning algorithm. In particular, we show several results (theorems) about our type-directed approach to pessimistic concurrency control of atomic transactions. The main result is that the execution of any well-typed program expressed in our language is guaranteed to satisfy the isolation property (defined formally in Section 2.2). We give a rigorous proof of isolation preservation and progress (up to deadlocks). The proof makes data accesses made by transactions explicit, and deals with multiple threads within an atomic transaction. This paper is a revised and extended version of (Woj05). An older version has also appeared in the author's habilitation thesis (Woj07).

1.1. *Basic Definitions and Motivating Example*

In the database community, atomic transactions are described by a set of properties referred to by the acronym ACID: Atomicity means that all changes to data are performed as if they are a single operation. That is, all the changes are performed, or none of them are. Consistency ensures that any transaction will take a database from one consistent state to another (this is a property that should be ensured by a program). Isolation means that the intermediate state of a transaction is invisible to other transactions. As a result, transactions that run concurrently appear to be serialized. Durability means that after a transaction successfully completes, changes to data persist and are not undone, even in the event of a system failure. In the programming language community, atomicity often implies isolation. A block of code that requires atomic execution is called a critical section. However, the "all-or-nothing" semantics is usually not guaranteed if a thread is killed in the middle of executing a critical section. Concurrent execution of critical sections (or atomic blocks) is isolated (or serialized). In this paper, we use the term isolation, meaning isolation as defined in databases and atomicity as defined in programming languages.

Let us begin from a small example. In Figure 1, we present a short program expressed using a ML-like programming language with `let`-binders and references, extended with software-based atomic transactions.

The example program consists of two concurrent parts that use the `atomic` construct to spawn two concurrent transactions A and B. They share two reference cells `a1` and `a2`, which have been created and initiated to 1000 using the `ref` construct. The `let` x `=` v `in` P construct from ML is used to bind a value v (here a reference cell) with a name x and continue with program P (x binds in P). Transaction A is performing a bank transfer—it withdraws 10 from an account `a1` and deposits 10 to an account `a2`. The accounts are implemented using reference cells. The 'read' expression $!x$ returns the current value stored in x, while the 'write' expression $x := v$ overwrites x with v. Transaction A also invokes a native method `print` of the operating system that prints out the state of accounts `a1` and `a2` before and after the update. Transaction B is computing the current balance `balance`, which is equal the total amount of assets deposited on accounts `a1` and

Shared data structures:

```
let a1 = ref 1000 in
let a2 = ref 1000 in
let balance = ref 0 in
```

Transaction A:

```
atomic (
  print a1;
  a1 := !a1 - 10;
  print a1, a2;
  a2 := !a2 + 10;
  print a2
)
```

Transaction B and continuation C:

```
atomic (
  balance := !a1 + !a2
);

let double = !balance + !balance in
print double
```

Figure 1. Atomic transactions A and B execute concurrently and exactly once.

`a2`. After the balance has been committed by transaction B, the program's continuation C prints out its double value.

A sequence of operations $e_1;...;e_n$ to be executed atomically by an atomic transaction, is declared using the construct `atomic` $(e_1;...;e_n)$. Isolation (or atomicity) means that transactions that run concurrently appear to be serialized and all transaction operations are executed exactly once (as are the operations of a critical section), unless an explicit rollback construct is invoked. Thus, a sequence of transaction operations can be regarded as a single unit of computation, regardless of any other operations occurring concurrently. So, the total balance is equal 2000, even if the concurrent 'read' and 'write' operations would be interleaved. The concurrent execution of code blocks A and B without `atomic` might give balance equal 1990.

To support concurrency on multicore CPUs, the implementation of `atomic` should allow transaction operations to be executed in parallel whenever possible. A single lock is therefore not suitable. An implementation of `atomic` using optimistic concurrency control will allow transactions to be executed in parallel and without blocking. If on transaction commit or earlier, the system would detect that some transaction (we call it a *conflicting* transaction) has read inconsistent state then the transaction is rolled back and reexecuted. Unfortunately, this behaviour invalidates the intended semantics of the `print` function call and any other methods whose effects are not easily revocable; such methods must be executed by a transaction exactly once. A workaround is to forbid the use of native methods inside transactions (e.g. (PJGF96)) or defer the execution of operations with irrevocable effects after transaction commit (e.g. (Har05)). Unfortunately, these simple techniques are not always plausible. For example, in our program the values printed by transaction A reflect intermediate states of transaction processing, so printing these values cannot be shifted before or after this transaction.

1.2. *Calculus Design*

In this paper, we define a calculus of atomic transactions relaying entirely on pessimistic concurrency control. In this approach, a transaction is never conflicting with other transactions, so no implicit rollback is required. The key idea of our design is to schedule the 'read' and 'write' operations of concurrent atomic transactions, so that the isolation (or atomicity) property is preserved and the transactions are executed exactly once. The scheduling algorithm may delay (temporarily block) the execution of the 'read' and 'write' operations that appear "too early". The decision whether to delay an operation on some object or not, is made by comparing versions associated with the object with versions hold by the atomic transaction. We can think of *versions* as integer values that are incremented after some actions have occurred. Each atomic transaction has to obtain a consistent snapshot of versions for all objects it may ever use before it is allowed to access any shared object for the first time. Since a version snapshot is guaranteed to be unique for all atomic transactions, it can therefore be used to obtain an exclusive access to the objects. Moreover since versions are ordered, atomic transactions can access the shared objects in the order which guarantees isolation (or atomicity). In our previous work (Woj07; WRS04), we have designed several such algorithms, varying in the level of concurrency and required input data.

Our calculus of atomic transactions can be seen as the core of an intermediate language for translation from the concrete syntax of a programming language used by programmers to the executable code. In Figure 2, we present the translation of our example program to the calculus. Below we briefly explain the language constructs used in this translation. The syntax and operational semantics of all constructs of our calculus will be given in Section 2. The details of the translation algorithm are beyond scope if this paper.

Execution of `newlock` $l : m$ `in` e creates a new unique name of a *versioning lock* l (or *verlock* in short) of type m to be used in program e. The type m is a *singleton* verlock type, i.e. the type of a single verlock. Both l and m may be referred to in the expression e, i.e. x and m are bound in e. A fresh verlock is created for every native method and a data structure that must be accessed atomically. In our example program, verlocks `l1`, `l2` and `l3` have been created for the reference cells respectively, `a1`, `a2` and `balance`. A reference creation $\texttt{ref}_m\, e$ is decorated with a singleton verlock type m (for some m).

Execution of `atomic` $\overline{e}$ e creates a new atomic transaction for the evaluation of expression e. Concurrent execution of atomic transactions satisfies the isolation property. After the creation, e commences execution, in parallel with the rest of the body of the spawning program. Each transaction is therefore executed by a new fresh thread. The $\overline{e}$ expression should give the input data for the concurrency controller. In the simplest case, it is a sequence of verlocks $l_1, ..., l_n$ that are used by an atomic transaction to guard critical operations, where the *critical operations* are all 'read' and 'write' operations on reference cells and other operations (or method calls) that must be executed by the transaction atomically.

Each critical operation of an atomic transaction is enclosed by the `sync` construct, which implements a *critical section* guarded by a verlock. The verlocks provide 'hooks' for the scheduling algorithm, which may then delay operations guarded by verlocks, so

Shared data structures:

```
newlock l1 : m in
newlock l2 : n in
newlock l3 : o in

let a1 = ref_m 1000 in
let a2 = ref_n 1000 in
let balance = ref_o 0 in
```

Transaction A:

```
atomic l1,l2 (
  print a1
  sync l1 a1 := !a1 - 10;
  print a1, a2;
  sync l2 a2 := !a2 + 10;
  print a2
)
```

Transactions B and C:

```
atomic l1,l2,l3 (
  sync l3 balance := sync l1 !a1
    + sync l2 !a2
);

atomic l3 (
  let double = sync l3 !balance
    + sync l3 !balance in
  print double
)
```

Figure 2. Concurrent execution of atomic transactions B and C is equivalent to a sequential execution of B followed by C.

that the execution of concurrent transactions is isolated. Other critical operations (not guarded by verlocks) are not isolated. This design decision allows isolation to be relaxed if needed, e.g. some input/output operations should never be blocked.

Evaluation of the `sync` e e' expression is as follows. The expression e is evaluated first, and should yield a verlock, which is then acquired when possible. The expression e' is then evaluated, giving a value v. Finally, the verlock is released and the value v is returned as the result of the whole expression. Note that the assignment expression `:=` enclosed in `sync` will return just an empty value. At first sight, the `sync` e e' expression is similar to the `synchronized` statement in Java (GJSB00). However, verlocks combine simple locks (mutexes) for protection against simultaneous access by concurrent threads, with a pessimistic concurrency control algorithm responsible for isolated execution of atomic transactions. An example algorithm will be described in Section 2.3.

Atomic transactions can be nested. If a transaction would be nested (or enclosed) in some other atomic transaction (let us call it *external*), then as long as they can acquire verlocks, they run in parallel. If the two transactions would like to acquire the same verlock, the scheduling algorithm ensures that the nested transaction will acquire this verlock after the external transaction has acquired the verlock and released it for the last time. Thus, in the (ideal) serial execution of the transactions, the nested transaction would commence after the external transaction has completed. It follows from the definition, that the isolation property of atomic transactions is satisfied.

In order to better utilize multicore CPU architectures, we require transactional code to be executed in parallel with the main program code whenever possible. To achieve this goal, each atomic transaction is executed by a separate thread and the continuation code of every atomic transaction (which is not explicitly declared as a transaction) is

translated at the level of the intermediate language to a single atomic transaction that can be executed in parallel with other transactions (see transaction C in Figure 2). Any transactions declared by the continuation code in the source program will be nested in the continuation transaction in the translated program. The concurrency control algorithm ensures, however, that the concurrent execution of all transactions is equivalent to a serial execution of these transactions. The order in this serial run agrees with the (partial) order defined by the source program.

Our language allows multithreaded programs by including the expression `fork` e, which spawns a new thread for the evaluation of expression e. This evaluation is performed only for its effect; the result of e is never used. Threads spawned by an atomic transaction are executed *within the scope* of this transaction. i.e. isolation (or atomicity) is defined with respect to complete code executed by the transaction, including any threads spawned by the transaction but excluding any nested transactions that these threads may spawn. An atomic transaction terminates when all its threads have completed. In general, an execution of a single multithreaded transaction may not be deterministic due to the interleaving of transaction internal threads. However, the isolation (or atomicity) property of transactions is defined with respect to whole transactions (encapsulating these internal threads). Thus, when talking about whole transactions we can still use the above definitions to describe behaviour of programs expressed in our calculus.

Our calculus also includes the `rollback` and `retry` language constructs. They can be used inside transactions to respectively, rollback and rollback-and-restart a transaction on demand, if some condition is not met. Transaction rollback increases language expressiveness, however, this feature is orthogonal to the design of a type system for safe pessimistic concurrency control, which is our primary focus in this paper. We therefore present the calculus without these constructs.

1.3. *Safe Concurrency Control*

The first argument of the `atomic` $\overline{e}$ e construct specifies the input data required for the scheduling algorithm to work correctly. Passing wrong data can jeopardize isolation. Thus, to make our language safe, we present in this paper a type system that can statically verify if the data passed to the scheduling algorithm will be correct. For instance, our example program does not typecheck if any argument of `atomic` (i.e. `l1`, `l2`, or `balance`) would be removed. A type-error also occurs when access to any reference cell (or method) decorated by some verlock type is not enclosed by the `sync` construct.

The type system verifies two conditions: (1) all 'read' and 'write' operations on reference cells are guarded by some verlock, and (2) a verlock being an argument of a critical section declared using `sync` is an argument of the `atomic` construct used to declare an atomic transaction enclosing this critical section. These two conditions are necessary to guarantee the correct execution of the scheduling algorithm used for pessimistic concurrency control. The type system builds on Flanagan and Abadi's (FA99) type system for detection of race conditions, which ensures that all accesses to shared data are protected by locks. We extended this simple locking principle to verify the above two conditions.

The type system could be refined if needed, so that operations having immutable effects could be left unguarded and thus invisible to the concurrency controller.

Isolation has been proposed as the correctness condition of concurrency control algorithms for atomic transactions (BHG87). The isolation guarantee in our language stems from three sources: (1) compile time enforcement that each shared data location (or an input/output operation) is guarded by a verlock and that threads acquire the corresponding verlock before accessing the location (or performing the input/output operation), (2) compile time enforcement that requires that all verlocks to be acquired during a transaction, are declared at the beginning of the transaction, and (3) a runtime locking strategy that assigns versions to transactions that allow them to acquire verlocks so that isolation is preserved.

Given the type system, it is easy to propose a simple but inefficient algorithm translating from the source language used to express our example program in Figure 1 to the calculus of atomic transactions: (1) annotate each new reference cell (and definition of a method) that must be executed atomically with a fresh verlock type; (2) create fresh verlocks, one per verlock type; (2) wrap occurrences of all critical operations with `sync`, using verlocks corresponding to verlock types introduced in step (1); and (3) assign an argument of every `atomic` construct, permuting over all declared verlocks and type-checking the program until the verification succeeds. The type-checking algorithm will be described in Section 3. We leave open the problem of finding a more efficient translation algorithm.

1.4. *Implementation*

To demonstrate usefulness of our approach, we developed *Atomic RMI* (ARM11)—an extension of Java Remote Method Invocation (RMI) (RMI) with support of distributed atomic transactions. Java RMI is a system for creating distributed applications, where methods of remote objects may be invoked from other Java Virtual Machines (JVMs), located on the same host or on different hosts. Java RMI marshals and unmarshals arguments and results of remotely-called methods retaining object-oriented polymorphism and type information. Our library provides constructs on top of Java RMI, allowing the programmer to declare a series of method calls on remote objects as a distributed atomic transaction. Such a transaction guarantees the properties of atomicity (either all of the operations of a transaction are performed or none), consistency (after any transaction finishes, the system remains in a valid—or consistent—state), and isolation (each transaction perceives itself as being the only currently running transaction).

Atomic RMI exercises pessimistic concurrency control using fine grained verlocks (a single verlock per remote object) while simultaneously providing support for rolling back transactions (using a `rollback` construct), and restarting them (using a `retry` construct). The two constructs increase expressiveness while retaining the advantages of pessimistic concurrency control, such as a free use of methods that may have irrevocable effects in transactions that do not use the `rollback` and `retry` constructs. The concurrency control protocol that we have implemented in Atomic RMI is the combination of the BVA versioning algorithm defined in this paper and SVA (Woj07) which allows

more concurrency than BVA; the algorithms have been extended to support transaction rollback and distribution. Versioning locks are built into the remote method invocation mechanism and completely transparent to the programmer. The version counters are kept as part of the object stubs which are used to handle calls on the objects.

Our system also includes *Atomic RMI Precompiler*—a tool which serves to help the users of Atomic RMI by automatically inferring upper bounds on the number of times each transactional remote object may be used in each transaction (such data are required by the SVA algorithm). It is a commandline utility which analyzes Java source files (or complete projects) relying on the Jimple intermediate language (VRH98) and, on the basis of the information collected during the analysis, generates some additional lines of code. This code specifies for each transaction which objects may be used within that transaction and up to how many times each of them may be expected to be invoked. These instructions are then inserted into the source code (either in-place, or into new files) before each transaction begins. If the upper bounds on object calls cannot be inferred for a given transaction, the transaction is executed using the BVA algorithm.

1.5. *Paper Structure*

The paper is organized as follows. Section 2—the heart of our paper—defines syntax, semantics, and typing of the calculus. Section 3 states and proves the main results of isolation preservation and type soundness. Section 4 shows and proves dynamic correctness of the BVA versioning algorithm. Section 5 discusses related work, and Section 6 concludes. Appendix A contains the standard part of proofs.

2. Calculus of Atomic Transactions

Below we define the core part of our language formally as a statically typed calculus, equipped with a structural operational semantics. The semantics has been split into a high-level semantics of the intermediate language, and a low-level semantics of an example concurrency control algorithm. We have used the semantics to formally prove correct the algorithm, and to show several results (theorems) about our type-directed approach to pessimistic concurrency control. The main result is that well-typed programs satisfy the isolation property. In the Appendix, we give a rigorous proof of isolation preservation and progress for our language (up to deadlocks); the proof makes data accesses explicit and deals with multiple threads within an atomic transaction. This is one of the first such proofs for atomic transactions.

2.1. *Syntax*

The calculus of atomic transactions is defined as the call-by-value λ-calculus, extended with atomic transactions and versioning locks. The abstract syntax is in Figure 3. The main syntactic categories are values and expressions. We write $\overline{x}$ as shorthand for a possibly empty sequence of variables $x_1, ..., x_n$ (and similarly for $\overline{t}$, $\overline{e}$, etc.).

$$
\begin{array}{lrll}
\text{Variables} & x, y & \in \mathit{Var} & \\
\text{Type Var-s} & m, o & \in \mathit{TypVar} & \\
\text{Allocations} & a, b & \in 2^{\mathit{TypVar}} & \\
\text{Permissions} & p & \in 2^{\mathit{TypVar}} & \\
\text{Types} & s, t & & ::= \texttt{Unit} \mid t \to^{a,p} t \mid \texttt{Ref}_m\, t \mid m \\
\text{Values} & v, w & \in \mathit{Val} & ::= () \mid \lambda^{a,p} x : t.\, e \\
\text{Expressions} & e & \in \mathit{Exp} & ::= x \mid v \mid e\ e \mid \texttt{ref}_m\, e \mid\ !e \\
 & & & \mid e := e \mid \texttt{newlock}\ x{:}m\ \texttt{in}\ e \mid \texttt{sync}\ e\ e \\
 & & & \mid \texttt{fork}\ e \mid \texttt{atomic}\ \overline{e}\ e
\end{array}
$$

We work up to alpha-conversion of expressions throughout, with x binding in e in expressions $\lambda x : t.\, e$.

Figure 3. The calculus of atomic transactions: Syntax

Types include the base type `Unit` of unit expressions, which abstracts away from concrete ground types for basic constants (integers, Booleans, etc.), the type $t \to^{a,p} t$ of functions, the type $\texttt{Ref}_m\, t$ of reference cells containing a value of type t, and finally a singleton verlock type m. The types of references and functions are decorated by correspondingly, m and a, p, where m is a singleton verlock type of a verlock used to protect the reference cell against simultaneous accesses by concurrent threads, and a and p describe an *allocation* and *permission*. Allocations and permissions are sets of singleton verlock types, representing respectively, the set of all verlocks that may be *demanded* during evaluation of a function, and the set of verlocks that must be *held* before a function call.

A value is either an empty value () of type `Unit`, or function abstraction $\lambda^{a,p} x : t.\, e$ (decorated with allocation a and permission p). Values are first-class programming objects, they can be passed as arguments to functions and returned as results and stored in reference cells. Basic expressions e are mostly standard and include variables, values, function applications, reference creation $\texttt{ref}_m\, e$ (decorated with a singleton verlock type m), and the usual imperative operations on references, i.e. dereference $!e$ and assignment $e := e$. We also assume existence of `let`-binders, and use syntactic sugar $e_1; e_2$ (sequential execution) for $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ (for some x, where x is fresh). The remaining expressions have already been explained in Section 1.2.

2.2. *Operational Semantics*

We specify the operational semantics using the rules defined in Figure 4, 5, and 6. The isolation (or atomicity) property is defined with respect to a program state. A state S (see Figure 4) consists of three elements: a verlock store π, a reference store σ, and a collection of expressions T, which are organized as a sequence $T_0, ..., T_n$. They first two compounds are sometimes referred to collectively as a store π, σ. Each expression T_i in the sequence represents a *concurrent thread*.

The *lock store* π is a finite map (or dictionary) from lock locations to their states. A verlock location has two states, unlocked (0) and locked (1), and is initially unlocked.

State Space

$$
\begin{array}{rcll}
S & \in & \mathit{State} & = & \mathit{LockStore} \times \mathit{RefStore} \times \mathit{ThreadSeq} \\
\pi & \in & \mathit{LockStore} & = & \mathit{LockLoc} \to \{0,1\} \\
\sigma & \in & \mathit{RefStore} & = & \mathit{RefLoc} \to \mathit{Val} \\
l & \in & \mathit{LockLoc} & \subset & \mathit{Var} \\
r & \in & \mathit{RefLoc} & \subset & \mathit{Var} \\
pv & \in & \mathit{VerMap} & \subset & \mathit{LockLoc} \to \mathbf{Nat} \\
gv & \in & \mathit{VerMap} & \subset & \mathit{LockLoc} \to \mathbf{Nat} \\
lv & \in & \mathit{VerMap} & \subset & \mathit{LockLoc} \to \mathbf{Nat} \\
T & \in & \mathit{ThreadSeq} & ::= & f \mid T, T \\
f & \in & \mathit{Exp}_{ext} & ::= & x \mid v \mid f\ e \mid v\ f \\
 & & & & \mid \mathtt{ref}_m\ f \mid\ !f \mid f := e \mid r := f \\
 & & & & \mid \mathtt{newlock}\ x{:}m\ \mathtt{in}\ e \mid \mathtt{sync}\ f\ e \mid \mathtt{insync}\ l\ f \\
 & & & & \mid \mathtt{fork}\ e \mid \mathtt{atomic}\ \overline{f}\overline{e}\ e \mid \mathtt{atomic}\ \overline{lf}\ e \mid \mathtt{transact}\ pv\ T
\end{array}
$$

Evaluation Contexts

$$
\begin{array}{rcl}
\mathcal{E} & = & [\,] \mid \mathcal{E}\ e \mid v\ \mathcal{E} \\
 & & \mid \mathtt{ref}_m\ \mathcal{E} \mid\ !\mathcal{E} \mid \mathcal{E} := e \mid r := \mathcal{E} \\
 & & \mid \mathtt{sync}\ \mathcal{E}\ e \mid \mathtt{insync}\ l\ \mathcal{E} \\
 & & \mid \mathtt{atomic}\ \overline{l}\mathcal{E}\overline{e}\ e \mid \mathtt{transact}\ pv\ \mathcal{E} \mid \mathcal{E}, T \mid T, \mathcal{E}
\end{array}
$$

Figure 4. Reduction semantics – Part I

The *reference store* σ is a finite map from reference locations to values stored in the references. Lock locations l and reference locations r are simply special kinds of variables that can be bound only by the respective stores.

The expressions f are written in the calculus presented in Section 2.1, extended with a new construct `transact` $pv\ T$. The construct is not part of the language to be used by programmers—it will be used later to explain semantics.

We define a small-step evaluation relation $\langle \pi, \sigma \mid e \rangle \longrightarrow \langle \pi', \sigma' \mid e' \rangle$, read "expression e reduces to expression e' in one step, with stores π, σ being transformed to π', σ'". We also use $\longrightarrow^*$ for a sequence of small-step reductions.

By *concurrent evaluation*, or *concurrent run*, we mean a sequence of small-step reductions in which the reduction steps can be taken by different threads with possible interleaving.

Reductions are defined using evaluation context $\mathcal{E}$ for expressions e and f. The evaluation context ensures that the left-outermost reduction is the only applicable reduction for each individual thread in the entire program. Context application is denoted by $[\,]$, as in $\mathcal{E}[\,e\,]$. Structural congruence rules allow us to simplify reduction rules by removing the context whenever possible.

The evaluation of a program e starts in an initial state with empty stores $(\emptyset, \emptyset)$ and with a single thread that evaluates the program's expression e. Evaluation then takes

Structural Congruence

$$T, T' \equiv T', T$$

$$T, () \equiv T$$

$$\frac{\langle \pi, \sigma \mid T \rangle \longrightarrow \langle \pi', \sigma' \mid T' \rangle}{\langle \pi, \sigma \mid \mathcal{E}[\,T\,] \rangle \longrightarrow \langle \pi', \sigma' \mid \mathcal{E}[\,T'\,] \rangle}$$

$$\frac{T \longrightarrow T'}{\langle \pi, \sigma \mid T \rangle \longrightarrow \langle \pi, \sigma \mid T' \rangle}$$

Transition Rules

$$\begin{array}{rcl} \mathit{eval} & \subseteq & \mathit{Exp} \times \mathit{Val} \\ \mathit{eval}(e, v_0) & \Leftrightarrow & \langle \emptyset, \emptyset \mid e \rangle \longrightarrow^* \langle \pi, \sigma \mid v_0, (), \cdots, () \rangle \end{array}$$

$$\lambda x.\, e\ v \longrightarrow e\{v/x\} \qquad \text{(R-App)}$$

$$\frac{r \notin \mathit{dom}(\sigma)}{\langle \pi, \sigma \mid \mathtt{ref}_m\, v \,\rangle \longrightarrow \langle \pi, (\sigma, r \mapsto v) \mid r \rangle} \qquad \text{(R-Ref)}$$

$$\langle \pi, \sigma \mid !r \rangle \longrightarrow \langle \pi, \sigma \mid v \rangle \quad \text{if} \quad \sigma(r) = v \qquad \text{(R-Deref)}$$

$$\langle \pi, \sigma \mid r := v \rangle \longrightarrow \langle \pi, \sigma[r \mapsto v] \mid () \rangle \qquad \text{(R-Assign)}$$

$$\mathcal{E}[\,\mathtt{fork}\ e\,] \longrightarrow \mathcal{E}[\,()\,], e \qquad \text{(R-Fork)}$$

$$v_i, v'_j \longrightarrow v_i \quad \text{if} \quad i < j \qquad \text{(R-Thread)}$$

Figure 5. Reduction semantics – Part II

place according to the transition rules in Figure 5 and 6. The evaluation terminates once all threads have been reduced to values, in which case the value v_0 of the initial, first thread T_0 is returned as the program's result (typing will ensure that other values are empty values). Subscripts in values reduced from threads denote the sequence number of the thread, i.e. v_i is reduced from i's thread, denoted T_i $(i = 0, 1..)$. The execution of threads can be arbitrarily interleaved. Since different interleavings may produce different results, the evaluator $\mathit{eval}(e, v_0)$ is therefore a relation, not a partial function.

2.2.1. *Functions, reference cells and threads* Below we describe evaluation reduction rules in Figure 5. These are the standard rules of a call-by-value λ-calculus (Plo75), extended with references and threads.

The rule (R-App) is a beta-reduction rule of the call-by-value λ-calculus. We write $\{v/x\}e$ to denote the capture-free substitution of v for x in the expression e.

$$\frac{pv = \mathcal{F}(\bar{l})}{\langle \pi, \sigma \mid \mathcal{E}[\,\texttt{atomic}\ \bar{l}\ e\,]\rangle \longrightarrow \langle \pi, \sigma \mid \mathcal{E}[\,()\,], \texttt{transact}\ pv\ e\rangle} \quad \text{(R-Transact)}$$

$$\texttt{transact}\ pv\ \mathcal{E}[\,\texttt{fork}\ e\,] \longrightarrow \texttt{transact}\ pv\ (\mathcal{E}[\,()\,], e) \quad \text{(R-Fork')}$$

$$\frac{pv(l)\ \text{OK} \quad \text{for all} \quad l \in dom(pv)}{\langle \pi, \sigma \mid \texttt{transact}\ pv\ v\rangle \longrightarrow \langle \pi, \sigma \mid ()\rangle} \quad \text{(R-Commit)}$$

$$\frac{\langle \pi, \sigma \mid \texttt{transact}\ pv\ e_1\rangle \longrightarrow^* \langle \pi', \sigma' \mid v_1\rangle}{\langle \pi, \sigma \mid \texttt{transact}\ pv\ e_1; \texttt{transact}\ pv'\ e_2\rangle \longrightarrow^* \langle \pi', \sigma' \mid v_1, \texttt{transact}\ pv'\ e_2\rangle} \quad \text{(R-Serial)}$$

$$\frac{\langle \pi, \sigma \mid \texttt{transact}\ pv\ e_1; \texttt{transact}\ pv'\ e_2\rangle \longrightarrow^* \langle \pi', \sigma' \mid v_1, v_2\rangle}{\langle \pi, \sigma \mid \texttt{transact}\ pv\ e_1, \texttt{transact}\ pv'\ e_2\rangle \longrightarrow^* \langle \pi', \sigma' \mid v_1, v_2\rangle} \quad \text{(R-Isolated)}$$

$$\frac{l \notin dom(\pi)}{\langle \pi, \sigma \mid \texttt{newlock}\ x{:}m\ \texttt{in}\ e\rangle \longrightarrow \langle (\pi, l \mapsto 0), \sigma \mid e\{l/x\}\{o_l/m\}\rangle} \quad \text{(R-Lock)}$$

$$\frac{\pi(l) = 0 \quad pv(l)\ \text{OK}}{\langle \pi, \sigma \mid \texttt{transact}\ pv\ \mathcal{E}[\,\texttt{sync}\ l\ e\,]\rangle \longrightarrow \langle \pi[l \mapsto 1], \sigma \mid \texttt{transact}\ pv\ \mathcal{E}[\,\texttt{insync}\ l\, e\,]\rangle} \quad \text{(R-Sync)}$$

$$\frac{\pi(l) = 1}{\langle \pi, \sigma \mid \texttt{insync}\ l\, v\rangle \longrightarrow \langle \pi[l \mapsto 0], \sigma \mid v\rangle} \quad \text{(R-InSync)}$$

Figure 6. Transition Rules – Part III

The rules (R-Ref), (R-Deref), and (R-Assign) correspondingly, create a new reference cell with a store location r initially containing v, read the current store value, and assign a new value to the store located by r. The notation $(\sigma, r \mapsto v)$ means "the store that maps r to v and maps all other locations to the same thing as σ". For instance, let us look at the rule (R-Assign). We use the notation $\sigma[r \mapsto v]$ to denote update of map σ at r to v. Note that the term resulting from this evaluation step is just $()$; the interesting result is the updated store. An expression f *accesses* a reference location r if there exists some evaluation context $\mathcal{E}$ such that $f = \mathcal{E}[\,!r\,]$ or $f = \mathcal{E}[\,r := v\,]$. (Note that both assign and dereference operations are non-commutative.)

Evaluation of expression `fork` e in (R-Fork) creates a new thread which evaluates e. The result of evaluating expression e is discarded by rule (R-Thread). A program *completes*, or *terminates*, if all its threads reduce to a value. By (R-Thread), values of more recent threads are ignored, so that eventually only the value of the first thread T_0 will be returned by a program.

Below we explain the rules in Figure 6. They are common for all versioning concurrency control algorithms, while rules that will be described later, define our example versioning algorithm.

2.2.2. *Transaction creation, termination, threading and isolation* Evaluation of a term `atomic` $\bar{l}$ e spawns a new *atomic transaction* `transact` pv e for isolated evaluation of expression e, where pv are data specific to this transaction that are used by the concurrency control algorithm. In case of an algorithm presented in this paper, it is a map of *private version counters*, initialized to 0. pv remains constant for the transaction's lifetime. The transaction will be evaluated by a new thread; see (R-Transact). Transactions can also spawn their own threads using `fork`; see (R-Fork').

An atomic transaction `transact` pv e has *completed* (or *terminated*) if expression e yields a value and the concurrency controller allows the transaction to commit (modelled as 'pv OK'). The whole expression reduces then to an empty value; see (R-Commit). Thus, atomic transactions (similarly to threads) are used only for their side-effects, which are in our case modifications to the store or values read from the store. In a full-size language, these operations can also be regarded as operations with input/output irrevocable effects (e.g. delivering/sending a message).

We say that a state S is *transaction-free* if it does not have a context $\mathcal{E}[\,\texttt{transact}\ pv\ T\,]$. Any transaction-free state is called a *result state*. The result states subsume data stored in all reference cells.

Two transactions are executed *serially* if one transaction commences after another one has completed; see (R-Serial). This rule is never executed by our abstract machine–it is only used to define the isolation property (below). By *serial evaluation*, or *serial run*, we mean evaluation, in which *all* transactions are executed serially. Note that a serial run is also concurrent since serialized transactions may be themselves multithreaded.

Concurrent evaluation of atomic transaction expressions (of the form `transact` pv e) satisfies the isolation property if the intermediate state of a transaction is invisible to other transactions. As a result, transactions that run concurrently appear to be serialized. This property is formalized by rule (R-Isolated): the concurrent execution of transactions evaluates to the values and the store, that can also be obtained by a serial evaluation of these transactions.

The isolation can be captured precisely using the notion of transaction noninterference, defined as follows.

Definition 1 (Noninterference) *Transactions in a concurrent run do not interfere (or satisfy the noninterference property) if there exists some ideal serial run R^{s} of these transactions (a transaction can appear in R^{s} only once), such that given any reference cell, the order of accessing this reference cell by transactions in the concurrent run (possibly several times) is the same as in R^{s}.*

Note that if some reference cell is accessed by a transaction several times, then no other concurrent non-interfering transaction is allowed to access the intermediate state of this reference cell. Otherwise, we would not be able to construct an equivalent serial run of such transactions. The reference cell can only be accessed by some another transaction after the former transaction has accessed this reference for the last time.

By *isolated evaluation* (or *isolated execution*) of an expression (containing some atomic

transactions) we mean any evaluation of this expression that satisfies the isolation property, defined as follows.

Definition 2 (Isolation Property) *Evaluation of an expression e satisfies the isolation property if all atomic transactions evaluated as part of e do not interfere. A given program has the isolation property if all its terminating evaluations satisfy this property.*

Our definition of isolation is more conservative than in database systems, since both read and write operations are always isolated. We do so since we would like to be able to isolate all operations with irrevocable effects, performing both an output and an input.

Note that the isolated evaluation of critical operations defined by an atomic transaction is *atomic* since they appear to other transactions as a single operation.

2.2.3. *Verlock creation, acquisition and release* The expression `newlock` $x : m$ `in` e (see rule (R-Lock)) dynamically creates a new verlock location l (with the initial state 0), extending a verlock store π accordingly, and replaces the occurrences of x in e with l. It also replaces occurrences of m in e with a type variable o_l that denotes the corresponding singleton verlock type. A verlock store π that binds a verlock's location l also implicitly binds the corresponding type variable o_l with kind `Lock`; the only value of o_l is l. Below we sometimes confuse a verlock and the verlock's location, where it is clear from the context what we mean.

A verlock location l is *free* if $\pi(l) = 0$, otherwise it is not free.

The expression `sync` $l\ e$ evaluated by an atomic transaction reduces to `insync` $l\, e$ (with $\pi(l)$ assigned a value 1), which means that verlock l has been *acquired*, if two conditions are met (see (R-Sync)): (1) the verlock is free and, (2) $pv(l)$ hold by the transaction allows it to safely proceed without invalidating isolation (modelled as 'pv OK'). If neither condition holds, the transaction's thread executing `sync` $l\ e'$ must wait (other threads of this transaction are however not blocked). The decision whether safely 'proceed' or 'wait' is made dynamically by the scheduling algorithm. The expression e enclosed by `insync` $l\, e'$ is then evaluated. The condition (1) is required just to avoid races inside a transaction, and could be removed if a transaction is single-threaded.

The `insync` $l\, e$ expression is evaluated until expression e reduces to a value v; see (R-InSync). Then the whole expression is replaced by v with $\pi(l)$ assigned a value 0, which means that verlock l has been *released* and becomes free.

Our language guarantees evaluation progress, i.e. each verlock requested by a transaction will be eventually acquired, if only transactions are themselves deadlock-free and terminate. We discuss the deadlock issue in Section 3.3, after explaining typing.

2.3. *Concurrency Controller*

The abstract machine of our language employs a runtime locking strategy for pessimistic concurrency control based on versioning. Below we describe the *Basic Versioning Algorithm (BVA)*—an example algorithm that implements this strategy. In order to keep the abstract machine simple, we have chosen one of the simplest algorithms possible.

Require:
  for each verlock name l **do**
    $gv_{l_i} \leftarrow 0$
    $lv_{l_i} \leftarrow 0$
  end for

Operation `atomic` $\bar{l}$ e:
  lock
    forall $l_i \in \bar{l}$
      $gv_{l_i} \leftarrow gv_{l_i} + 1$
      $pv_k(l) \leftarrow gv_{l_i}$
    end for
  end lock
  execute `transact` pv_k e

Operation `sync` l e:
  wait l is free and $pv_k(l) - 1 = lv_l$
  execute e

Operation `transact` pv_k v:
  for all $l_i \in dom(pv_k)$ **do**
    wait $pv_k[l] - 1 = lv_{l_i}$
    $lv_{l_i} = lv_{l_i} + 1$
  end for

Figure 7. The Basic Versioning Algorithm for ensuring isolation of atomic transactions

We need to define some data structures that are required by the algorithm. Firstly, the program state S is extended with a map gv of *global version counters* $gv(l)$ for each verlock l in π (initialized to 0), where a *version* is a natural number playing a rôle of access capability. Secondly, each verlock l maintains a local version counter $lv(l)$, which is also initialized to 0. A map lv of *local version counters* is part of the state S, too. We write gv_l and lv_l as shorthand for $gv(l)$ and $lv(l)$. For clarity we usually omit the counters in the rules when possible. The BVA algorithm maintains an invariant that a local version of each verlock is equal or less than a global version of the verlock, and it is equal or greater than zero. Each atomic transaction holds a map pv which associates verlocks with globally unique versions. The map pv is created for a given set of verlocks atomically, and remains constant for the transaction's lifetime.

The algorithm is given by the following set of steps (see also pseudocode in Figure 7):

BVA-0: Upon verlock creation, initialize global and local counters for this new verlock to zero.

BVA-1: At the moment of spawning a new atomic transaction k using `atomic` $\bar{l}$ e, for each verlock $l_i \in \bar{l}$, where $i = 1, .., |\bar{l}|$, increment counter gv_{l_i} by one. Create a fresh read-only map (dictionary) pv_k that contains bindings from the locks l_i to their upgraded versions gv_{l_i}. Upgrading the version counters gv_{l_i} and creation of the transaction's private copy pv_k of the upgraded versions is an atomic operation.

BVA-2: An atomic transaction k can acquire a verlock l using `sync` l e guarding a critical operation e, only when two conditions are met: the verlock is free and the transaction holds a version of this verlock that matches the current local version lv_l of l, i.e.

$$pv_k(l) - 1 = lv_l \quad . \tag{1}$$

Otherwise, the verlock acquisition operation is pending. Checking the two conditions is an atomic operation. If the condition (1) is satisfied then we say that the transaction is *safe* to acquire the verlock l, i.e. accessing data guarded by this verlock does not invalidate isolation. Otherwise, the verlock l is not acquired even if it is free, thus blocking the transaction's thread (any other threads are not blocked).

BVA-3: After an atomic transaction `transact` pv_k e has completed its execution, i.e. all threads of this transaction have terminated, for each verlock $l_i \in dom(pv_k)$ in parallel, wait until condition (1) is true, then increment the local version of each verlock l_i, so that we have $lv_{l_i} = pv_k(l_i)$.

The BVA algorithm guarantees the isolated evaluation of transactions: a transaction is verlocked if it tries to access a reference cell accessed by some transaction that has not terminated yet. The verlock will be released after the latter transaction completes. This concurrency controller does not allow for much concurrency. In (Woj07), we describe the SVA and RVA algorithms that can–whenever possible–release the blocked transactions earlier, and so they permit more concurrency in the system but they are more complex.

We can define the BVA algorithm precisely using the operational semantics of our language. The reduction rules corresponding to the four steps of the algorithm are given in Figure 8. They can be seen as rules of an example abstract machine implementing the semantic rules (R-Lock), (R-Transact), (R-Sync), and (R-Commit) in Figure 6.

The rules of the abstract machine are mostly straightforward. Some explanation requires the rule (R-Commit). A terminating transaction must wait till condition (1) is true before it is allowed to increment local versions lv of verlocks and terminate. This is required since some verlocks declared by a transaction may never be used, e.g. if some branch of code has not been executed. If the condition (1) is not true for these versions, then it means that they are held by some other (older) concurrent transactions that may use them to access shared data. Thus, they cannot be incremented yet.

2.3.1. *Correctness assumptions* The BVA algorithm guarantees noninterference, provided the following two conditions hold. Firstly, programs do not have race conditions, i.e. no data can be accessed without first acquiring a verlock. Secondly, all verlocks that *may* (not necessarily have to) be used by a transaction are known at a time when the transaction is spawned, so that the (R-Transact) rule can create the private version for each such verlock type, stored in the transaction's map pv. To maximize parallelism, we require only such verlocks to be declared. In Section 2.4 we define typing rules intended for checking statically if these two conditions hold in programs expressed using our language. Then, we show in Section 3 that our type system is sound, making the language *safe by construction*.

2.4. *Static Typing*

The type system is formulated in Figure 9 as a deductive proof system, defined using conclusions (or judgments) and the static inference rules for reasoning about the judgments. The typing judgment for expressions has the form $\Gamma; a; p \vdash e : t$, read "expression e has type t in environment Γ with allocation a and permission p", where an environment Γ is a finite mapping from free variables to types. An expression e is a *well-typed program* if it is closed and it has a type t in the empty type environment, written $\vdash e : t$.

Our intend is that, if the judgment $E; a; p \vdash e : t$ holds, then any terminating execution of expression e is race-free, satisfies the isolation property, and yields values of type t, provided:

$$gv \in \mathit{VerMap} \subset \mathit{LockLoc} \to \mathbf{Nat}$$

$$lv \in \mathit{VerMap} \subset \mathit{LockLoc} \to \mathbf{Nat}$$

$$\mathit{eval}(e, v_0) \Leftrightarrow \langle \emptyset, \emptyset, \emptyset, \emptyset \mid e \rangle \longrightarrow^* \langle \pi, \sigma, gv, lv \mid v_0, (), \cdots, () \rangle$$

$$\frac{\begin{array}{c}\pi(l) \in \{0,1\} \\ 0 \leq lv(l) \leq gv(l) \quad \text{for all} \quad l \in \mathit{dom}(\pi)\end{array}}{\langle \pi, \sigma, gv, lv \mid e \rangle \longrightarrow \langle \pi', \sigma', gv', lv' \mid e' \rangle} \qquad \text{(Invar)}$$

$$\mathit{BVA\text{-}0}: \quad \frac{\begin{array}{c} l \notin \mathit{dom}(\pi) \\ gv' = (gv, l \mapsto 0) \quad lv' = (lv, l \mapsto 0)\end{array}}{\begin{array}{l}\langle \pi, \sigma, gv, lv \mid \texttt{newlock}\ x\!:\!m\ \texttt{in}\ e \rangle \longrightarrow \\ \langle (\pi, l \mapsto 0), \sigma, gv', lv' \mid e\{l/x\}\{o_l/m\} \rangle\end{array}} \qquad \text{(R-Lock)}$$

$$\mathit{BVA\text{-}1}: \quad \frac{\begin{array}{c}\bar{l} = l_1, ..., l_n \\ gv' = gv[l_i \mapsto gv(l_i) + 1] \quad i = 1..n \\ pv = (l_1 \mapsto gv'(l_1), ..., l_n \mapsto gv'(l_n))\end{array}}{\begin{array}{l}\langle \pi, \sigma, gv, lv \mid \mathcal{E}[\,\texttt{atomic}\ \bar{l}\ e\,] \rangle \longrightarrow \\ \langle \pi, \sigma, gv', lv \mid \mathcal{E}[\,()\,], \texttt{transact}\ pv\ e \rangle\end{array}} \qquad \text{(R-Transact)}$$

$$\mathit{BVA\text{-}2}: \quad \frac{\pi(l) = 0 \quad pv(l) - 1 = lv(l)}{\begin{array}{l}\langle \pi, \sigma, gv, lv \mid \texttt{transact}\ pv\ \mathcal{E}[\,\texttt{sync}\ l\ e\,] \rangle \longrightarrow \\ \langle \pi[l \mapsto 1], \sigma, gv, lv \mid \texttt{transact}\ pv\ \mathcal{E}[\,\texttt{insync}\ l\, e\,] \rangle\end{array}} \qquad \text{(R-Sync)}$$

$$\mathit{BVA\text{-}3}: \quad \frac{\begin{array}{c} pv(l) - 1 = lv(l) \\ lv' = lv[l \mapsto pv(l)] \quad \text{for all} \quad l \in \mathit{dom}(pv)\end{array}}{\langle \pi, \sigma, gv, lv \mid \texttt{transact}\ pv\ v \rangle \longrightarrow \langle \pi, \sigma, gv, lv' \mid () \rangle} \qquad \text{(R-Commit)}$$

Figure 8. Transition Rules of Basic Versioning Algorithm

(i) the thread executing e holds at least versioning locks (verlocks) described by permission p (Condition 1),

(ii) if expression e is part of a transaction, then the transaction has declared all verlocks described by allocation a (Condition 2), and

(iii) the free variables of e are given bindings consistent with Γ.

We will show in Section 3 that the type system is sound. Based on this result, we state dynamic correctness of our example concurrency control algorithm, which together with type soundness gives the expected result of isolation preservation.

Our type system is an extension of Flanagan and Abadi's type system for detecting race

Judgments

$\Gamma \vdash \diamond$	Γ is a well-formed typing environment
$\Gamma \vdash t$	t is a well-formed type in Γ
$\Gamma \vdash a, p$	a, p is a well-formed resource allocation and permission in Γ
$\Gamma; a; p \vdash e : t$	e is a well-typed expression of type t in Γ with allocation a and permission p

Typing Rules

$$\frac{}{\emptyset \vdash \diamond} \quad \text{(Env-}\emptyset\text{)}$$

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

$$\frac{\Gamma \vdash \diamond \quad m \notin dom(\Gamma)}{\Gamma, m :: \texttt{Lock} \vdash \diamond} \quad \text{(Env-}m\text{)}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{Unit}} \quad \text{(Type-Unit)}$$

$$\frac{\Gamma \vdash t \quad \Gamma \vdash t' \quad \Gamma \vdash a, p}{\Gamma \vdash t \to^{a,p} t'} \quad \text{(Type-Fun)}$$

$$\frac{\Gamma \vdash t \quad \Gamma \vdash m}{\Gamma \vdash \texttt{Ref}_m\, t} \quad \text{(Type-Ref)}$$

$$\frac{m :: \texttt{Lock} \in \Gamma}{\Gamma \vdash m} \quad \text{(Type-Lock)}$$

$$\frac{\Gamma \vdash \diamond \quad \Gamma \vdash m \quad \text{for all} \quad m \in a \cup p}{\Gamma \vdash a, p} \quad \text{(Alloc)}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma; a; p \vdash () : \texttt{Unit}} \quad \text{(T-Unit)}$$

$$\frac{x : t \in \Gamma}{\Gamma; a; p \vdash x : t} \quad \text{(T-Var)}$$

$$\frac{\Gamma, x : s; a; p \vdash e : t}{\Gamma; a'; p' \vdash \lambda^{a,p} x : s.\, e : s \to^{a,p} t} \quad \text{(T-Fun)}$$

$$\frac{\Gamma; a; p \vdash e : s \to^{a',p'} t \quad \Gamma; a; p \vdash e' : s \quad a' \subseteq a \quad p' \subseteq p}{\Gamma; a; p \vdash e\ e' : t} \quad \text{(T-App)}$$

$$\frac{\Gamma \vdash m \quad \Gamma; a; p \vdash e : t}{\Gamma; a; p \vdash \texttt{ref}_m\, e : \texttt{Ref}_m\, t} \quad \text{(T-Ref)}$$

$$\frac{\Gamma; a; p \vdash e : \texttt{Ref}_m\, t \quad m \in p}{\Gamma; a; p \vdash !e : t} \quad \text{(T-Deref)}$$

$$\frac{\Gamma; a; p \vdash e : \texttt{Ref}_m\, t \quad \Gamma; a; p \vdash e' : t \quad m \in p}{\Gamma; a; p \vdash e := e' : \texttt{Unit}} \quad \text{(T-Assign)}$$

$$\frac{\Gamma, m :: \texttt{Lock}, x : m; a; p \vdash e : t \quad \Gamma \vdash a, p \quad \Gamma \vdash t}{\Gamma; a; p \vdash \texttt{newlock}\ x : m\ \texttt{in}\ e : t} \quad \text{(T-Lock)}$$

$$\frac{\Gamma; a; p \vdash e : m \quad m \in a \quad \Gamma; a; p \cup \{m\} \vdash e' : t}{\Gamma; a; p \vdash \texttt{sync}\ e\ e' : t} \quad \text{(T-Sync)}$$

$$\frac{\Gamma; a; \emptyset \vdash e : \texttt{Unit}}{\Gamma; a; p \vdash \texttt{fork}\ e : \texttt{Unit}} \quad \text{(T-Fork)}$$

$$\frac{\Gamma; a; p \vdash e_i : m_i \quad \text{for all} \quad i = 1..|\overline{e}| \quad \Gamma; \{m_1\} \cup ... \cup \{m_{|\overline{e}|}\}; \emptyset \vdash e_0 : t}{\Gamma; a; p \vdash \texttt{atomic}\ \overline{e}\ e_0 : \texttt{Unit}} \quad \text{(T-Atomic)}$$

Figure 9. The first-order type system for the *iso*-calculus

conditions (FA99). It provides rules for proving that the above two conditions are always true for well-typed programs. Condition 1 is verified using an approach described in (FA99). The set of typing rules in Figure 9 has been obtained by extending this approach with allocations (needed to verify Condition 2), and adding rules for typing constructs of our calculus. Most of the typing rules are fairly straightforward. For simplicity, we present a first-order type system and omit subtyping of allocations. The subtyping rules would be similar to the subtyping rules in (FA99), where also extensions with polymorphism and existential types have been described.

To verify Conditions 1 and 2, a verlock l is represented at the type level with a singleton verlock type m that contains l. The singleton type allows typing rules to assert that a thread holds verlock l by referring to that type rather than to the verlock l. During typechecking, each expression is evaluated in the context of allocations a and permissions p. Including a singleton verlock type in the allocation a, respectively permission p, amounts to assuming that the corresponding verlock's version, respectively the corresponding verlock, are held during the evaluation of e.

For instance, consider typing dereference and assignment operations on references, as part of typechecking some expression e''. The corresponding rules (T-Deref) and (T-Assign) check if a singleton verlock type m decorating the reference type is among verlock types mentioned in the current permission p. The permission p can be extended with m only when typechecking a synchronization expression `sync` e e'', where e has type m (see typing of e in (T-Sync)).

To verify if a transaction e_0 executing `sync` e e' has declared verlock e of some type m, we introduce an allocation a and require that m is mentioned in a. Note that m can be added to allocation a only while typechecking the construct `atomic` that has spawned transaction e_0. The rule (T-Isol) creates the allocation a from singleton types of all verlocks declared by the transaction; the allocation is then used for typechecking the body of the transaction.

An allocation a and permission p decorate a function type and function definition, representing respectively, allocation a—the set of all verlocks that may be requested while evaluating the function and any thread spawned by it, and permission p—the set of verlocks that must be held before a function call. Note that allocations are preserved by thread spawning since we allow transactions to be multithreaded, while permissions are nulled since spawned threads do not inherit verlocks from their parent thread.

Rules (T-Fork) and (T-Isol) require the type of the whole expression to be `Unit`; this is correct since threads are evaluated only for their side-effects.

3. Well-typed Programs Satisfy Isolation

The fundamental property of the type system and abstract machine of our language is that evaluation of well-typed, terminating programs satisfies isolation. The first component of the proof of this property is a *type preservation* result, stating that typing is preserved during evaluation. The second one is a *progress* result, stating that evaluation of an expression never enters into a state for which there is no evaluation rule defined. To prove both results, we extended typing judgments from expressions *Exp* to expressions

Judgments

$\vdash S : t$ $\quad$ S is a well-typed state of type t

Rules

$$\frac{\Sigma(l) = \{0,1\} \quad \Sigma(o_l) = \texttt{Lock}}{\Sigma \mid \Gamma; a; p \vdash l : o_l} \quad \text{(T-LockLoc)}$$

$$\frac{\Gamma \vdash m \quad \Sigma(r) = t}{\Sigma \mid \Gamma; a; p \vdash r : \texttt{Ref}_m\, t} \quad \text{(T-RefLoc)}$$

$$\frac{\begin{array}{c} dom(\pi) = \{l_1, ..., l_j\} \quad dom(\sigma) = \{r_1, ..., r_k\} \\ \Sigma = l_1 : \{0,1\}, ..., l_j : \{0,1\}, r_1 : s_1, ..., r_k : s_k, \\ o_{l_1} :: \texttt{Lock}, ..., o_{l_j} :: \texttt{Lock} \\ |T| > 0 \quad \Sigma \mid \Gamma; a_i; p_i \vdash T_i : t_i \quad \text{for all } \; i < |T| \end{array}}{\vdash \langle \pi, \sigma \mid T \rangle : t_0} \quad \text{(T-State)}$$

$$\frac{\vdash S : t_0 \quad \vdash S' : t_0}{\vdash S + S' : t_0} \quad \text{(T-Choice)}$$

$$\frac{\begin{array}{c} \Sigma \mid \Gamma; a; p \vdash f_i : t_i \\ \Sigma \mid \Gamma; a'; p' \vdash f'_j : t_j \quad i < j \end{array}}{\Sigma \mid \Gamma; a; p \vdash f_i, f'_j : t_i} \quad \text{(T-Thread)}$$

$$\frac{\begin{array}{c} a = \{o_{l_1}, ..., o_{l_n}\} \quad \Sigma \mid \Gamma; a; p \vdash l_i : o_{l_i} \\ \Sigma \mid \Gamma; a; p \vdash pv(l_i) : \mathbf{Nat} \quad \text{for all } \; i = 1..n \\ \Sigma \mid \Gamma; a; p \vdash T : t \end{array}}{\Sigma \mid \Gamma; a; p \vdash \texttt{transact}\ pv\ T : \texttt{Unit}} \quad \text{(T-Transact)}$$

$$\frac{\begin{array}{c} \Sigma \mid \Gamma; a; p \vdash l : m \\ \Sigma \mid \Gamma; a; p \vdash f : t \quad m \in a \quad m \in p \end{array}}{\Sigma \mid \Gamma; a; p \vdash \texttt{insync}\ l\ f : t} \quad \text{(T-InSync)}$$

$$\mathbf{Nat} = 0, 1, 2, .. \;\; \text{(includes zero)}$$

Figure 10. Additional judgments and rules for typing states

Exp_{ext}, and then to states as shown in Figure 10. The judgment $\vdash S : t$ says that "S is a well-typed state yielding values of type t". We assume a single, definite type for every location in the store π, σ. These types have been collected as a *store typing* Σ—a finite function mapping locations to types, and type variables to kinds.

Type preservation and progress yield that our type system is *sound*, i.e. it guarantees that if a program is well-typed then:

(*i*) each operation on references requires to first obtain a verlock, and
(*ii*) if obtaining a verlock is part of some transaction spawned using the `atomic` construct, then the transaction has a private version of this verlock (which is possible only if the name of it is the argument of the construct).

The first property is called *absence of race conditions* and is guaranteed by Abadi and Flanagan's type system for avoiding race conditions that we have extended. The second property is called *absence of non-declared verlocks* and is guaranteed by our extension of their type system. Based on the two properties of the type system, we have proven that evaluation of well-typed, terminating programs satisfies the isolation property. Below we give some nonstandard or more interesting parts of the proof; the rest is in the Appendix A.

Below we state formally the absence of race conditions and the absence of non-declared verlocks properties. Next, we give our main result of isolation preservation in Section 3.3.

3.1. *Absence of Races*

After removing allocations a and the rule (T-Isol) for typing the construct `atomic` in Figure 9, and replacing the semantics of verlocks by simple locks, we obtain Flanagan and Abadi's first-order type system (FA99). The fundamental property of this type system is that well-typed programs do not have race conditions. Below are three Lemmas and one Theorem as found in (FA99), with locks replaced by verlocks and extended with store typing Σ and allocations that appear in our language. (We quote these lemmas and the theorem since they will be used in our proof of type soundness.) We give them without any proofs since our extensions do not invalidate the original proof of race-freedom.

The semantics can be used to formalize the notion of a race condition, as follows. A state has a *race condition* if its thread sequence contains two expressions that access the same reference location. A program e has a race condition if its evaluation may yield a state with a race condition, i.e. if there exists a state S such that $\langle \emptyset, \emptyset \mid e \rangle \longrightarrow^* S$ and S has a race condition.

Independently of the type system, verlocks provide mutual exclusion, in that two threads can never be in a critical section on the same verlock. An expression f is in a *critical section* on a verlock location l if $f = \mathcal{E}[\,\texttt{insync}\ l\ f'\,]$ for some evaluation context $\mathcal{E}$ and expression f'. The judgment $\vdash_{cs} S$ says that at most one thread is in a critical section on each verlock in S. According to Lemma 1, the property $\vdash_{cs} S$ is maintained during evaluation.

Lemma 1 (Mutual Exclusion)
If $\vdash_{cs} S$ *and* $S \longrightarrow S'$, *then* $\vdash_{cs} S'$.

Lemma 2 says that a well-typed thread accesses a reference cell only when it holds the protecting verlock.

Lemma 2 (Lock-Based Protection)
Suppose that $\Sigma \mid \Gamma; a; p \vdash f : t$*, and* f *accesses reference location* r*. Then* $\Sigma \mid \Gamma; a; p \vdash r : \mathtt{Ref}_m\, t'$ *for some verlock type* m *and type* t'*. Furthermore, there exists verlock location* l *such that* $\Sigma \mid \Gamma; a; p \vdash l : m$ *and* f *is in a critical section on* l*.*

The lemma below implies that states that are well-typed and well-formed with respect to critical sections do not have race conditions.

Lemma 3 (Race-Conditions-Free States) *Suppose* $\vdash S : t$ *and* $\vdash_{cs} S$*. Then* S *does not have a race condition.*

Finally, we can conclude that well-typed programs do not have race conditions.

Theorem 1 (Absence of Race Conditions)
If $\vdash e : t$ *then* e *does not have a race condition.*

3.2. *Absence of Non-declared Verlocks*

Let us first define the notions used to state the properties of our type system (see the beginning of Section 3). An expression f is *part of* an atomic transaction `transact` $pv\ T$ if $T = \mathcal{E}[\,f\,]$ for some evaluation context $\mathcal{E}$. A transaction `transact` $pv\ T$ *has a version* of a verlock l if $pv(l)$ is defined. An expression f *has a version* of a verlock l if there exists some transaction which has a version of l, and f is part of this transaction. An expression f *requests* a verlock location l if $f = \mathcal{E}[\,\mathtt{sync}\ l\ e\,]$ for some evaluation context $\mathcal{E}$ and expression e. An atomic transaction `transact` $pv\ T$ is in a *critical section* on a verlock location l, if some thread of T is in a critical section on the verlock location l.

Now, for the complete language with `atomic` and `transact`, the judgment $\vdash_{cs} S$ says in addition to mutual exclusion property stated in Section 3.1, that each transaction being in a critical section on some verlock in state S has a version of this verlock (see Figure 11). According to Lemma 4, the property $\vdash_{cs} S$ is maintained during evaluation.

Lemma 4 (Version-Completeness Preservation) *If* $\vdash_{cs} S$ *and* $S \longrightarrow S'$*, then* $\vdash_{cs} S'$*.*

Proof. State S may consist of several threads that are evaluated concurrently. Suppose $S = \pi, \sigma \mid \mathcal{E}[\,\mathtt{transact}\ pv\ T\,]$ for some well-typed store π, σ, context $\mathcal{E}$ and (possibly multithreaded) term T. By rule (R-Commit) and evaluation context for `transact`, we know that transaction `transact` $pv\ T$ can either reduce to the empty value $()$ if T is a value, or to `transact` $pv\ T'$ otherwise, where T' is some expression. The former case is trivial since we have immediately

$$\emptyset \vdash_{cs} () \tag{2}$$

Judgments

$\mathcal{M} \vdash_{cs} f$	f has exactly one critical section for each verlock in $\mathcal{M}$
$\mathcal{M} \vdash_{cs} \texttt{transact}\ pv\ T$	transaction T has a version $pv(l)$ for each verlock l in $\mathcal{M}$
$\vdash_{cs} S$	S is well-formed with respect to critical sections and transactions
$\vdash_{tf} S$	S is well-formed and transaction-free

Rules for Critical Sections

$$\frac{f = x \mid v \mid \texttt{newlock}\ x{:}m\ \texttt{in}\ e \mid \texttt{fork}\ e}{\emptyset \vdash_{cs} f} \quad \text{(CS-Empty)}$$

$$\frac{\begin{array}{c}\mathcal{M} \vdash_{cs} f \\ f' = f\ e \mid v\ f \mid \texttt{ref}_m\ f \mid\ !f \\ \mid f := e \mid r := f \mid \texttt{sync}\ f\ e\end{array}}{\mathcal{M} \vdash_{cs} f'} \quad \text{(CS-Exp)}$$

$$\frac{\mathcal{M} \vdash_{cs} f}{\mathcal{M} \uplus \{l\} \vdash_{cs} \texttt{insync}\ l\ f} \quad \text{(CS-InSync)}$$

$$\frac{\begin{array}{c}\forall i < |T| \quad \mathcal{M}_i \vdash_{cs} T_i \\ \mathcal{M} = \mathcal{M}_0 \uplus \ldots \uplus \mathcal{M}_{|T|-1} \\ \forall l \in \mathcal{M} \quad \pi(l) = 1\end{array}}{\vdash_{cs} \langle \pi, \sigma \mid T \rangle} \quad \text{(CS-State)}$$

$$\frac{\begin{array}{c}\forall i = 1..|\overline{f}| \quad \mathcal{M}_i \vdash_{cs} f_i \\ \mathcal{M} = \mathcal{M}_1 \uplus \ldots \uplus \mathcal{M}_{|\overline{f}|} \\ f' = \texttt{atomic}\ \overline{f}\ e\end{array}}{\mathcal{M} \vdash_{cs} f'} \quad \text{(CS-Isol)}$$

$$\frac{\begin{array}{c}\forall i < |T| \quad \mathcal{M}_i \vdash_{cs} T_i \\ \mathcal{M} = \mathcal{M}_0 \uplus \ldots \uplus \mathcal{M}_{|T|-1} \\ \forall l \in \mathcal{M} \quad pv(l) \ \text{ is defined and } \ pv(l) > 0\end{array}}{\mathcal{M} \vdash_{cs} \texttt{transact}\ pv\ T} \quad \text{(CS-Task)}$$

$$\frac{\begin{array}{c}\vdash_{cs} \langle \pi, \sigma \mid T \rangle \\ \forall i < |T| \quad T_i \neq \texttt{transact}\ pv\ T'\end{array}}{\vdash_{tf} \langle \pi, \sigma \mid T \rangle} \quad \text{(TF-State)}$$

Figure 11. Judgments and rules for reasoning about critical sections and transaction free states

by (CS-Empty), which is what we needed.

Let us now consider the latter case. Suppose that transaction `transact` pv T is in a critical section on some verlock location l. From premise $\vdash_{cs} S$, we have

$$\mathcal{M} \vdash_{cs} \texttt{transact}\ pv\ T \tag{3}$$

for some $\mathcal{M}$ by (CS-State) and the fact that transaction `transact` pv T is a thread in S (by (R-Transact)). But then by (CS-Task)

$$l \in \mathcal{M} \tag{4}$$

and version $pv(l)$ is defined. Now we need to consider two subcases, depending on if the reduction step of T enters a new critical section, or not.

Case a). Reduction to a new critical section.
Consider an evaluation step from T to T', such that T has `sync` l' e in its redex position. Thus, by rule (R-Sync) $T' = \mathcal{E}'[\,\texttt{insync}\ l'\,e\,]$ and $\pi(l') = 1$ for some context $\mathcal{E}'$, lock location l', and expression e, where $l' \neq l$. Hence, T' is in a critical section on verlock l'. Note that by mutual exclusion (Lemma 1) it is not possible to have a reduction step from T to T' if $l' = l$ since (3) and (4) hold.

Let us assume that `transact` pv T' does not have a version of verlock l', i.e. $pv(l')$ is not defined. But this is not possible, since by version-based protection Lemma 5 (below), if a transaction `transact` pv T requests verlock location l', then version $pv(l')$ is defined, which contradicts our assumption (since we also know that the private versions map pv is preserved by the reduction step as it is never modified). Thus, $\mathcal{M}' \vdash_{cs} \texttt{transact}\ pv\ T'$ and precisely $\mathcal{M}' = \mathcal{M} \uplus \{l'\}$ by (CS-InSync). From the latter, we have $l \in \mathcal{M}'$ by (4).

Case b). No new critical section.
Consider reduction from T to T' such that T has in its redex position an expression other then `sync` l' e. But then from (3) we have $\mathcal{M} \vdash_{cs} \texttt{transact}\ pv\ T'$ since T' is in the same critical sections as T, and we know that $l \in \mathcal{M}$ and $pv(l)$ is defined.

From (2), a) and b) we obtain the needed result $\vdash_{cs} S'$ by type preservation Corollary 2 (in Section A.1.1) and (CS-State) and induction on threads in S. □

Lemma 5 says that a well-typed thread obtains a verlock only when it holds a version of this verlock.

Lemma 5 (Version-Based Protection) *Suppose that* $\Sigma \mid \Gamma; a; p \vdash f : t$*, and* f *requests a verlock location* l*. Then* $\Sigma \mid \Gamma; a; p \vdash l : m$ *for some verlock type* m*. Furthermore, there exists a transaction* `transact` pv T *which* f *is part of, such that* $\Sigma \mid \Gamma; a; p \vdash$ `transact` pv T : `Unit` *and version* $pv(l)$ *is defined.*

Proof. If f requests a verlock location l then from the definition of "requesting a verlock location" we have $f = \mathcal{E}[\,\texttt{sync}\ l\ e'\,]$ for some evaluation context $\mathcal{E}$ and expression e'. Suppose that $\Sigma \mid \Gamma; a; p \vdash \texttt{sync}\ l\ e' : t'$ for some type t'. Then, by (T-Sync) we have

$$\Sigma \mid \Gamma; a; p \vdash l : m \tag{5}$$

for some verlock type m, and $m \in a$. From the latter and premise $\Sigma \mid \Gamma; a; p \vdash f : t$, we know that f must be part of some transaction with allocation a (since $a \neq \emptyset$).

Hence, by (T-Isol) f is reduced from some expression `atomic` $\bar{l}$ e_0, such that $\Sigma \mid \Gamma; a'; p' \vdash$ `atomic` $\bar{l}$ e_0 : `Unit` (for some a' and p'), where $\bar{l}$ is a sequence of verlock locations. Moreover, since allocation a is preserved during transaction evaluation (since only (T-Isol) can modify a) we have $\Sigma \mid \Gamma; a; \emptyset \vdash e_0 : t''$ for some t'', also by (T-Isol).

From the above, we have immediately $l \in \bar{l}$ by (5) and (T-Isol) since $m \in a$. (Note that (T-Isol) is the *only* rule which could add m to allocation a.)

But then, by (R-Transact) expression e_0 can only reduce to `transact` pv e_0 for some pv, such that version $pv(l)$ is defined, which is precisely the needed result since pv is constant and so it does not change while expression e_0 would reduce to T such that $T = \mathcal{E}'[\, f \,]$ for some context $\mathcal{E}'$. By (T-Transact), term `transact` pv T has type `Unit`, which completes the proof. □

Note that the above property implies that in our language all verlock requests are part of some transaction. This feature has simplified the type system and reasoning about the isolation property. A full-size language could make a difference between accessing a verlock as part of some transaction, or outside transactions.

We conclude that all verlocks used by each transaction in well-typed programs are known a priori.

Theorem 2 (Verlock-Usage Predictability) *All verlocks that may be requested by an atomic transaction of a well-typed program are known before the transaction begins.*

Proof. By lock-based protection Lemma 2, it is enough to show that the argument $\bar{l}$ of the `atomic` $\bar{l}$ e construct used to spawn a transaction, is a sequence of all verlocks that *may* be requested by the transaction. The proof is straightforward by the version-based protection Lemma 5, version-completeness preservation Lemma 4, and induction on transactions and verlock location requests. □

The above result implies that the BVA algorithm will be able to create upon a transaction's creation, a private version of each verlock that may be used by the transaction.

3.3. *The Main Result of Isolation Preservation*

We have defined the isolated evaluation for complete transactions (see Section 2.3). This is however not a problem since in practice we are interested only in result states of this evaluation. Below we therefore formulate an isolation preservation result for traces (i.e. sequences of evaluated states) that begin and finish in a transaction-free state. The judgment for such states has the form $\vdash_{tf} S$, read "state S is well-formed and transaction-free", which means that either no transaction has been spawned yet, or if there were any, then they have already completed.

Below we state that each trace of a well-typed program has the "isolation up to" property, provided that the corresponding evaluation finishes in a result state.

Lemma 6 (Isolation Property Up To) *Suppose* $\Sigma \mid \emptyset;\emptyset;\emptyset \vdash S : t$ *and* $\vdash_{tf} S$. *If* $S \longrightarrow^* S'$ *and* $\vdash_{tf} S'$, *then the run* $S \longrightarrow^* S'$ *satisfies the isolation property up to* S'.

Proof. From premise $\vdash_{tf} S$, we have $\vdash_{cs} S$ by (TF-State). From the latter and premise $\Sigma \mid \emptyset;\emptyset;\emptyset \vdash S : t$, each transaction in S (if we would let S not to be transaction-free) is well-typed by (T-State), and by version-based protection Lemma 5, it has versions of all verlocks it may request. Moreover, by version-completeness preservation Lemma 4, we know that this property is preserved by reduction from S to S'' for some state S''. Hence, it is also preserved by any following reductions up to S' (by re-applying Lemma 4). Thus, it holds in all states reached by any transactions that could be spawned by these reductions. But this is precisely one of the two requirements for the correctness of the "isolated evaluation" using the BVA algorithm (i.e. Property 2, see **??**).

Moreover, from $\vdash_{cs} S$, the lock-based protection Lemma 2 and mutual exclusion Lemma 1 give another requirement (i.e. Property 1, see Section **??**) for the correctness of evaluation using the BVA algorithm.

By premises $\vdash_{tf} S$ and $\vdash_{tf} S'$, we also know that the evaluation has begun and finished with no active transactions. Hence, by dynamic correctness of the BVA algorithm, stated by the noninterference Theorem 6 (the theorem and the proof will be given in Section 4) and the definition of isolation, we obtain the needed result. □

We say that a program is *terminating* if all its runs terminate; a run *terminates* if it reduces to a value. Based on the above lemma, we can prove that well-typed, terminating programs satisfy the isolation property:

Theorem 3 (Isolation Property) *If* $\vdash e : t$, *then all terminating runs* $e \longrightarrow^* v_0$, *where* v_0 *is some value of type* t, *satisfy the isolation property.*

Proof. From premise $\vdash e : t$, e is a closed, well-typed term. Consider any well-typed store π, σ, that is $\Sigma \mid \emptyset;\emptyset;\emptyset \vdash \pi, \sigma$ for some Σ. Then $\vdash \pi, \sigma \mid e : t$ by Definition 6 (see Section A.1.1) and (T-State). Moreover, we have

$$\vdash_{tf} \pi, \sigma \mid e \tag{6}$$

since program e (before commencing its execution) does not have any transaction by syntax (see Figure 3). Pick up any terminating trace such that $\pi, \sigma \mid e \longrightarrow^* \pi', \sigma' \mid v_0$ for some store π', σ' and value v_0. From (6), we have $\vdash_{cs} \pi', \sigma' \mid v_0$ by (TF-State) and version-completeness preservation (Lemma 4). From the latter, and the fact that $v_0 \neq$ `transact` $pv\ T$ for any pv and T, we get $\vdash_{tf} \pi', \sigma' \mid v_0$, which together with (6) implies that the run satisfies the isolation property up to v_0 by Lemma 6. Then the result follows by induction on the length of the terminating reduction sequences from $\pi, \sigma \mid e$ to any value. □

3.3.1. *Deadlocks* We stated our main result for terminating programs. Note however that if a program deadlocks or never terminates, all its runs reaching some result state have the "isolation up to" property (up to this state). Thus, the deadlock issue is orthogonal to the goals of our work, and can be solved using the existing approaches.

The only deadlocks possible in our language stem from either two threads of the same transaction trying to acquire two verlocks l_1 and l_2 in parallel but in a different order, or when a thread tries to acquire a verlock again before releasing it. This means however that other transactions that want to acquire these verlocks will be also blocked. Deadlock can be avoided by imposing a strict partial order on verlocks within each transaction, and respecting this order when acquiring verlocks; our language and type system can be extended with this principle by embodying the solution described in (FA99).

Some thread systems (e.g. C# and Java) implement "re-entrant locking": from within a "lock" statement the program can call another of its methods that also locks the same object, with no risk of deadlock. However, the feature is double-edged (Bir05) since the other method may be called at a time when the monitor invariants are not true, leading to misbehavior. Our system prohibits re-entrant locking and such misbehavior is prevented, being replaced by a deadlock.

3.4. *Type Soundness*

Reduction of a program may either continue forever, or may reach a final state, where no further evaluation is possible. Such a final state represents either an answer or a type error. Since programs expressed in our language are not guaranteed to be deadlock-free (unless all transactions are single-threaded and re-entrant locking is allowed), we also admit a deadlocked state to be an (acceptable) answer. Thus, proving type soundness means that well-typed programs yield only well-typed answers.

Our proof of type soundness rests upon the notion of type preservation (also known as subject reduction). The type preservation property states that reductions preserve the type of expressions. Type preservation by itself is not sufficient for type soundness. In addition, we must prove that programs containing type errors are not typable. We call such expressions with type errors *faulty expressions* and prove that faulty expressions cannot be typed. Below are only the main results. A proof of type safety and evaluation progress can be found in the Appendix A.

3.4.1. *Type safety*

The statement of the main type preservation lemma must take stores and store typings into account. For this we need to relate stores with assumptions about the types of the values in the stores. Below we define what it means for a store π, σ to be well typed. (For clarity, we omit permissions p from the context and global gv and local lv counters from states when possible.)

Definition 3 *A store π, σ is said to be* well typed *with respect to a store typing Σ and a typing context Γ, written $\Sigma \mid \Gamma; a \vdash \pi, \sigma$, if $dom(\pi, \sigma) = dom(\Sigma)$ and $\Sigma \mid \Gamma; a \vdash \mu(l) : \Sigma(l)$ for every store $\mu \in \{\pi, \sigma\}$ and every $l \in dom(\mu)$.*

Intuitively, a store π, σ is consistent with a store typing Σ if every value in the store has the type predicted by the store typing.

By canonical forms (Lemma 31 in Section A.1.2), each *location value* $l \in dom(\pi, \sigma)$ can be either a verlock location, or a reference location, depending on a concrete type.

For simplicity, we often refer to π, σ as the store, meaning individual stores, i.e. either π or σ, depending on a given value and type. If a location value l is a verlock location then it is kept in a verlock store π; if the value is a reference location then it is kept in a reference store σ.

Type preservation for our language states that the reductions defined in Figures 4, 5 and 8 preserve type:

Theorem 4 (Type Preservation) *If* $\Sigma \mid \Gamma; a \vdash T : t$ *and* $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ *and* $\langle \pi, \sigma \mid T \rangle \longrightarrow \langle (\pi, \sigma)' \mid T' \rangle$*, then for some* $\Sigma' \supseteq \Sigma$, $\Sigma' \mid \Gamma; a \vdash T' : t$ *and* $\Sigma' \mid \Gamma; a \vdash (\pi, \sigma)'$.

The type preservation theorem asserts that there is some store typing $\Sigma' \supseteq \Sigma$ (i.e., agreeing with Σ on the values of all the old locations) such that a new term T' is well typed with respect to Σ'. This new store typing Σ' is either Σ or it is exactly $(\Sigma, l : t_0)$, where l is a newly allocated location, i.e. the new element of $dom((\pi, \sigma)')$, and t_0 is the type of the initial value bound to l in the extended store $(\mu, l \mapsto v_0)$ for some $\mu \in \{\pi, \sigma\}$.

Proof. The proof is a straightforward induction on a derivation of $T : t$, using the lemmas below and the inversion property of the typing rules. The proof proceeds by case analysis according to the reduction $T \longrightarrow T'$. See Appendix A for the proof. □

A key lemma that we use in the proof of type preservation is the replacement lemma. It allows the replacement of one of the subexpressions of a typable expression with another subexpression of the same type, without disturbing the type of the overall expression.

Lemma 7 (Replacement) *If:*

1 $\mathcal{D}$ *is a deduction concluding* $\Sigma \mid \Gamma; a \vdash \mathcal{E}[\, e_1 \,] : t$,
2 $\mathcal{D}'$ *is a subdeduction of* $\mathcal{D}$ *concluding* $\Sigma' \mid \Gamma'; a' \vdash e_1 : t'$,
3 $\mathcal{D}'$ *occurs in* $\mathcal{D}$ *in the position corresponding to the hole* $(\mathcal{E})$ *in* $\mathcal{E}[\,]$, *and*
4 $\Sigma' \mid \Gamma'; a' \vdash e_2 : t'$

then $\Sigma \mid \Gamma; a \vdash \mathcal{E}[\, e_2 \,] : t$.

Proof. See a proof in (WF94) for a language with no stores and store typing; the proof is also valid for our language. □

The substitution lemma is the key to showing type preservation for reductions involving substitution.

Lemma 8 (Substitution) *If* $\Sigma \mid (\Gamma, x : t); a \vdash e : t'$ *and* $\Sigma \mid \Gamma; a \vdash v : t$*, then* $\Sigma \mid \Gamma; a \vdash e\{v/x\} : t'$.

Proof. We proceed by induction on a derivation of the statement $(\Gamma, x : t) \vdash e : t'$, and case analysis on the final typing rule used in the proof. (For clarity, we remove store typing Σ, allocation and permission whenever possible.) See Appendix A for the proof. □

A corollary of Type Preservation (Theorem 7) is that reduction steps preserve type.

Corollary 1 (Type Preservation) *If* $\Sigma \mid \Gamma; a \vdash T : t$ *and* $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ *and* $\langle \pi, \sigma \mid T \rangle \longrightarrow^* \langle (\pi, \sigma)' \mid T' \rangle$*, then for some* $\Sigma' \supseteq \Sigma$*,* $\Sigma' \mid \Gamma; a \vdash T' : t$ *and* $\Sigma' \mid \Gamma; a \vdash (\pi, \sigma)'$*.*

Proof. If $\langle \pi, \sigma \mid T \rangle \longrightarrow \langle (\pi, \sigma)' \mid T' \rangle$, then $T = \mathcal{E}[\, e_1 \,]$ and $T' = \mathcal{E}[\, e_2 \,]$, and $\langle \pi, \sigma \mid e_1 \rangle \longrightarrow \langle (\pi, \sigma)' \mid e_2 \rangle$ and $\Sigma' \mid \Gamma; a \vdash (\pi, \sigma)'$, for some $\Sigma' \supseteq \Sigma$, so $\Sigma' \mid \Gamma; a \vdash T' : t$ by the replacement Lemma 26. Then the result follows by induction on the length of the reduction sequence $\langle \pi, \sigma \mid T \rangle \longrightarrow^* \langle (\pi, \sigma)' \mid T' \rangle$. □

3.4.2. *Evaluation progress* Subject reduction ensures that if we start with a typable expression, then we cannot reach an untypable expression through any sequence of reductions. This by itself, however, does not yield type soundness.

Below we prove that evaluation of a typable expression cannot get *stuck*, i.e. either the expression is a value or there is some reduction defined. However, we do allow reduction to be suspended indefinitely since our language is not deadlock-free. This is acceptable since we define and guarantee isolation, respectively isolation-up-to, only for programs that either terminate, or reach some result state (see Theorem 3 and Lemma 6).

A canonical forms lemma states the possible shapes of values of various types.

Lemma 9 (Canonical Forms)
1) If v *is a value of type* `Unit`*, then* v *is* $()$*.*
2) If v *is a value of type* $t \to^{a,p} s$*, then* $v = \lambda^{a,p} x : t.\, e$*.*
3) If v *is a value of type* m*, then* v *is a verlock location.*
5) If v *is a value of type* $\texttt{Ref}_m\, t$*, then* v *is a reference cell location (or reference location, in short) of a reference cell storing values of type* t*.*

Proof. Straightforward from the grammar in Figure 3 and the extended grammar in Figure 4. □

We state progress only for closed expressions, i.e. with no free variables. For open terms, the progress theorem fails. This is however not a problem since complete programs—which are the expressions we actually care about evaluating—are always closed.

Independently of the type system and store typing, we should define which state we regard as well-formed. Intuitively, a state is well-formed if the content of the store is consistent with the expression executed by the thread sequence. (We omit global and local counters that are also part of the state, as they are not represented in expressions explicitly.) In case of store π, if there is some evaluation context $\mathcal{E}[\, \texttt{insync}\ l\, e \,]$ in the thread sequence for any verlock location l, then $\pi(l)$ should contain 1, marking that the verlock has been acquired. As for the store σ, containing the content of each reference cell, we may only require that it is well typed.

Definition 4 *Suppose* π, σ *is a well-typed store, and* $\overline{f}$ *is a well-typed sequence of expressions, where each expression is evaluated by a thread. Then, a state* $\pi, \sigma \mid \overline{f}$ *is* well-formed*, denoted* $\vdash_{wf} \pi, \sigma \mid \overline{f}$*, if for each expression* f_i *(*$i < |\overline{f}|$*) such that* $f_i = \mathcal{E}[\, \texttt{insync}\ l\, e \,]$ *for some* l*, there is* $\pi(l) = 1$*.*

Of course, a well-typed, closed expression with empty store is well-formed.

According to Lemma 32, the property $\vdash_{wf} \pi, \sigma \mid \overline{f}$ is maintained during evaluation.

Lemma 10 (Well-Formedness Preservation) *If* $\vdash_{wf} \pi, \sigma \mid \overline{f}$ *and* $\pi, \sigma \mid \overline{f} \longrightarrow (\pi, \sigma)' \mid \overline{f}'$ *then* $\vdash_{wf} (\pi, \sigma)' \mid \overline{f}'$.

Proof. Consider a well-formed state $\pi, \sigma \mid e_0$, for some well-typed program $\vdash e_0 : t$ and well-typed store π, σ. Suppose that $e_0 = \mathcal{E}[\,\texttt{sync}\ l\ e\,]$ for some context $\mathcal{E}$, and $\pi(l) = 0$. (Note that when a verlock location l is created, then initially $\pi(l) = 0$ by (R-Lock).) From the latter and premise that the state is well-formed, we know that there is no context $\mathcal{E}'$ such that $e_0 = \mathcal{E}'[\,\texttt{insync}\ l\, e'\,]$ for any e'. From the latter and premise, by (R-Sync), we could reduce expression e_0 to $(\pi, \sigma)' \mid e_1$, such that $e_1 = \mathcal{E}[\,\texttt{insync}\ l\, e\,]$. But then, after reduction step, we have $\pi(l) = 1$ (again by (R-Sync)). Moreover, by type preservation Theorem 7, the new state is well typed. Thus, from the definition of well-formedness, we get immediately that $\vdash_{wf} (\pi, \sigma)' \mid e_1$. Finally, we obtain the needed result by induction on thread creation. □

A state $\pi, \sigma \mid T$ is *deadlocked* if there exist only evaluation contexts $\mathcal{E}$, such that $T = \mathcal{E}[\,\texttt{sync}\ l\ e\,]$ for some verlocks l, such that $\pi(l) = 1$ for each l (i.e. the verlocks are not free) and there is no other evaluation context possible.

Now, we can state the progress theorem.

Theorem 5 (Progress) *Suppose T is a closed, well-typed term (that is, $\Sigma \mid \emptyset; \emptyset; \emptyset \vdash T : t$ for some t and Σ). Then either T is a value or else, for any store π, σ such that $\Sigma \mid \emptyset; \emptyset; \emptyset \vdash \pi, \sigma$ and $\vdash_{wf} \pi, \sigma \mid T$, there is some term T' and store $(\pi, \sigma)'$ with $\pi, \sigma \mid T \longrightarrow (\pi, \sigma)' \mid T'$, or else T is deadlocked on some verlock(s).*

Proof. Straightforward induction on typing derivations. We need only show that either $\pi, \sigma \mid T \longrightarrow (\pi, \sigma)' \mid T'$, or T is a value, or $\pi, \sigma \mid T$ is a deadlocked state. From the definition of $\longrightarrow$, we have $T \longrightarrow T'$ iif $T = \mathcal{E}[\,e_1\,]$, $T' = \mathcal{E}[\,e_1'\,]$, and $e_1 \longrightarrow e_1'$.

Case The variable case cannot occur (because e is closed).
Case The abstract case is immediate, since abstractions are values.
Case $T = e_1\ e_2$ with $\vdash e_1 : t \to^{b,p} s$ and $\vdash e_2 : t$
By the induction hypothesis, either e_1 is a value or else it can make a step of evaluation, and likewise e_2, or T is a deadlocked state. If e_1 can take a step, then $e_1 = \mathcal{E}_1[\,e'\,]$ and $e' \longrightarrow e''$. But then $T = \mathcal{E}[\,e'\,]$ where $\mathcal{E} = \mathcal{E}_1\ e_2$; thus $T \longrightarrow \mathcal{E}[\,e''\,]$. Otherwise, e_1 is a value. If e_1 is a value and e_2 can take a step, then $e_2 = \mathcal{E}_2[\,e'\,]$ and $e' \longrightarrow e''$ then $T = \mathcal{E}[\,e'\,]$ where $\mathcal{E} = e_1\ \mathcal{E}_2$; thus $T \longrightarrow \mathcal{E}[\,e''\,]$. Otherwise, e_1 and e_2 are values, or T is a deadlocked state. Finally, if both e_1 and e_2 are values, then the canonical forms lemma tells us that e_1 has the form $\lambda^{b,p} x : t.\, e_1'$, and so rule (R-App) applies to T.

Other cases are straightforward induction on typing derivations, following the pattern of the case with $T = e_1\ e_2$. □

4. Dynamic Correctness of Basic Versioning

Independently of the type system, we must prove that our example versioning algorithm (BVA) used for scheduling of transaction operations is correct, i.e. it can be used to evaluate programs so that all possible executions satisfy the isolation property.

The BVA algorithm is correct only for programs that have the following two properties:

Property 1 *All data accesses are protected by verlocks.*

Property 2 *Each transaction has a version of each verlock it may use.*

But these two properties correspond precisely to the absence of race freedom, and the absence of undeclared verlocks properties. We have shown that they hold for all well-typed programs (see Theorems 1 and 2). Thus, to prove the correctness of the BVA algorithm, it remains to show that all transactions of a well-typed program never interfere (from the definition of isolation).

From the definition of `sync` $l\ e$, we know that a locked expression e can be executed only by a single thread since other threads would be blocked (due to the atomicity property of verlocks). Moreover, by the absence of race conditions Theorem 1, we know that in order to access a reference, first a verlock must be taken. Therefore, we can formulate the definition of noninterference using verlocks instead of references:

Definition 5 (Noninterference (Verlocks)) *Tasks in a concurrent run* do not interfere *(or satisfy the* noninterference *property) if there exists some ideal serial run* R^{s} *of all these transactions, such that given any verlock, the order of acquiring the verlock by transactions in the concurrent run is the same as in* R^{s}*.*

Below we prove the BVA algorithm, given by evaluation rules *BVA-0-3* in Figure 8. We require steps *BVA-1* and *BVA-2* to be atomic. We write gv_l and lv_l as shorthand for $gv(l)$ and $lv(l)$.

Essentially, the BVA algorithm implements ordering of verlock acquisitions based on versions. Tasks acquire verlocks in such order as is required to satisfy the noninterference property. We need to show that all possible evaluations of a typable expression cannot lead to a transaction-free state that is not obtainable by some serialized evaluation of transactions. Note that we do not require a program to terminate. However, we consider its correctness only for a set of transactions that will eventually terminate.

To prove the correctness of the algorithm, we only need to show that all transactions of each well-typed program never interfere (from the definition of isolation).

The proof proceeds by proving lemmas about safety and liveness properties of verlocks, verlock-based mutual exclusion, and finally about ordering properties of verlock-based access to references. We begin from introducing a few definitions.

For a transaction `transact` $pv\ e$ where $pv(l)$ is defined, we define *access* of this transaction to a verlock l, denoted a, as a pair $(pv(l), lv_l)$, where $pv(l)$ and lv_l are correspondingly, a private and local versions of verlock l. Access of `transact` $pv\ e$ to a verlock l is *defined* if $pv(l)$ is defined.

Access $a_k = (pv_k(l), lv_l)$ of a transaction k is *valid* if condition (1) is true. A transaction *gets* a valid access $(pv_k(l), lv_l)$ when condition (1) is becoming true.

4.1. *Verlock Access*

Lemma 11 (Verlock Safety) *A verlock can be acquired only by a transaction which has valid access to the verlock.*

Proof. Straightforward from the definition of access and the premise of (R-Sync). □

Lemma 12 (Access Liveness) *Each access of a given transaction in a concurrent run will be eventually valid, provided that all transactions terminate.*

Proof. Let k_0 be the first transaction, with access a_{k_0} to some verlock l defined. By steps *BVA-0* and *BVA-1*, $a_{k_0} = (pv_{k_0}(l), lv_l)$, where $pv_{k_0}(l) = 1$ and $lv_l = 0$. Moreover, access a_{k_0} is valid since condition (1) is true. Consider a transaction k_1 created after k_0, with access a_{k_1} to l defined, where $a_{k_1} = (2, 0)$. The access a_{k_1} is not valid since (1) is false ($2 - 1 \neq 0$). However, since we assumed that transactions terminate, then by step *BVA-3*, the local version of verlock l will be eventually upgraded by 1 as soon as k_0 terminate. But then a_{k_1} is valid. Hence, by induction on transactions, we will get the needed result. □

Lemma 13 (Verlock Liveness) *Each non-free verlock requested by a transaction will be eventually acquired, provided that it will be released.*

Proof. Straightforward from access liveness Lemma 12 and the premise of (R-Sync). □

Lemma 14 (Private-Version Uniqueness) *Each transaction has a unique private version of each verlock during transaction lifetime.*

Proof. Immediate from step *BVA-1*, where for each verlock l, $pv(l)$ is given a value equal gv_l increased by one, and the fact that step *BVA-1* is atomic and $pv(l)$ is constant. □

Lemma 15 (Access Uniqueness) *For each verlock and any transaction which has access to this verlock defined, the access is globally unique.*

Proof. Immediate from the definition of access and the private version uniqueness Lemma 14. □

Lemma 16 (Valid-Access Mutual Exclusion) *At any time, there is only one access to a given verlock which is valid.*

Proof. Consider a verlock l. Since local version lv_l of this verlock is the same for all transactions at any time, from private-version uniqueness Lemma 14, we have that at any given time, there is only one transaction which can have access for which validity condition (1) is true. Hence, we obtain the needed result. □

Lemma 17 (Access Privacy) *A valid access a_k of a transaction k can be invalidated only by transaction k.*

Proof. Consider a valid access $a_k = (pv_k(l), lv_l)$ of some transaction k to a verlock l. By access uniqueness Lemma 15, there is no other transaction k' with access $(pv_{k'}(l), lv_l)$ such that $pv_{k'}(l) = pv_k(l)$. On the other hand, from valid-access mutual exclusion Lemma 16, we know that it is not possible that some other transaction could have (different) access to verlock l that is also valid. Thus, we know that only k has a valid access to l. Moreover, by step *BVA-3* we know that transaction k can only upgrade lv_l if (1) is true. It means that lv_l can only be upgraded if k has a valid access a_k to l. But this is precisely the needed result, since by modifying lv_l access a_k to l is no longer valid. □

Lemma 18 (Valid-Access Preservation) *If a transaction has got valid access to a verlock, then it will have valid access to it at any time (until it would invalidate it).*

Proof. Straightforward from valid-access mutual exclusion Lemma 16 and access privacy Lemma 17. □

Lemma 19 (Verlock-Set Mutual Exclusion) *As long as a transaction is allowed to acquire a verlock l, no other transaction can acquire verlock l.*

Proof. Straightforward from valid-access-preservation Lemma 18 and verlock safety Lemma 11. □

By verlock-set mutual exclusion Lemma 19, and the fact that we are not interested in the relative order of lock acquisitions made by the same transaction (since *any* such order would satisfy Definition 5 of noninterference), we can represent all acquisitions of a given verlock made by a given transaction by any single such acquisition. Thus, in the rest of the proof, we can consider a system in which each verlock is acquired by a transaction at most once. By Lemma 19, the proven result will be valid for any system.

4.2. *Access Ordering*

Lemma 20 (Access Ordering) *The order of acquiring a verlock by transactions corresponds to the order in which transactions got valid access to it.*

Proof. Immediate by verlock safety Lemma 11 and verlock-set mutual exclusion Lemma 19. □

Lemma 21 (Valid-Access Ordering) *The relative order of getting valid access to a verlock by transactions corresponds to the order of creating the transactions.*

Proof. Consider a transaction k, which gets valid access to some verlock l. Access becomes valid when condition (1) becomes true. By step *BVA-3*, this occurs when some other transaction k' upgrades a local version lv_l by 1. By access privacy and valid-access mutual exclusion, the transaction k' has valid access to l and is the only one which has

it. The valid access of k' becomes invalidated after upgrading lv_l by 1, and then given to k. From the latter and (1), we can derive that

$$pv_{k'}(l) = pv_k(l) - 1 \ . \tag{7}$$

Moreover, from step *BVA-1*, we know that the order of private versions corresponds to the order of creating transactions, i.e. if k_i has been created before k_j, then $pv_{k_i}(l) < pv_{k_j}(l)$ for each verlock l such that both transactions have defined access to it. Hence, from (7), we know that k' has been created before k. Finally, by induction on transactions we obtain the needed result. □

Lemma 22 (Total Ordering) *The relative order of acquiring a verlock by transactions is the same for every verlock.*

Proof. Immediate from verlock safety Lemma 11, verlock-set mutual exclusion Lemma 19, and access ordering Lemma 20, valid-access ordering Lemma 21, and the fact that the order of creating transactions is total (by step *BVA-1*). □

Lemma 23 (Natural Ordering) *The order of acquiring verlocks by transactions in a concurrent run is the same as in some serial run.*

Proof. By the definition of a serial run of transactions, we have immediately that all verlocks are acquired by the transactions in the order in which the transactions have been created (let's call this property a "natural order").

From verlock safety Lemma 11, valid-access ordering Lemma 21, verlock-set mutual exclusion Lemma 19, and total ordering Lemma 22, it is straightforward that any concurrent run has the "natural order" property. Moreover, since we only consider isolation for expressions that reached a transaction-free state (see Lemma 6), hence we are allowed to consider only concurrent runs in which all transactions terminate. This means that each verlock acquired must be eventually released (note that all verlocks are initially free by (R-Lock)). Thus, by verlock liveness Lemma 13, all verlocks requested will be eventually acquired. From the latter, we conclude that there can be a plausible serial run considered, and obtain the needed result. □

4.3. *Noninterference*

We can now state the main result about the BVA algorithm:

Theorem 6 (Noninterference) *If a given program has Properties 1 and 2, then any evaluation of this program up to any result state, using the BVA concurrency control algorithm, satisfies the noninterference property.*

Proof. By natural ordering Lemma 23, the noninterference property is satisfied in any concurrent run in which verlocks are acquired when permitted by the algorithm, which completes the proof. □

By the noninterference theorem and definition of isolation property in Section 2.2.2, we conclude that the BVA algorithm can be used to implement the isolated execution of transactions.

5. Related Work

The work in this paper builds on research on atomic transactions (or atomicity) in three areas: programming language design and verification, formal semantics, and concurrency control. We give some examples below.

There have been a lot of proposals of extending programming languages with support of atomicity or atomic transactions. The authors in (HKM$^+$94; WFMN92) proposed an extension of the ML programming language with atomic transactions, which can be composed by the programmer using higher-order functions, each function implementing one transactional feature. Thus, transactions can be declared to satisfy only a subset of the atomicity, isolation, and durability features. Transactions can be multithreaded, similarly to our atomic transactions.

In recent years, there has been a growing interest in adopting atomic transactions to general-purpose programming languages. These efforts are centered around the notion of *Software Transactional Memory (STM)* (ST95). STM allows the declaration of atomic transactions for controlling access to shared memory in concurrent computing. It can be regarded as an alternative to lock-based synchronization. The STM mechanism is similar to database transactions but it operates on (volatile) memory and it is incorporated into a general-purpose programming language. For the last few years, many object-level and word-level STMs have been developed for different programming languages (see e.g. (HF03; HMPH05; RG05; NWAT$^+$08; HLMS03; HLM06) among others).

For instance, Harris and Fraser (HF03) Extented Java with *Conditional Critical Regions (CCRs)* (Hoa72). The programmer can guard a conditional region by an arbitrary boolean condition, with calling threads blocking until the guard is satisfied. It is also possible to explicitly terminate and rollback an execution of an atomic block, if some condition is not met. The implementation is based on mapping CCRs onto a word-level STM, which groups together series of memory accesses and makes them appear atomic. Unlike our language, atomic transactions are executed optimistically. This restricts the availability of I/O operations within an atomic block.

Despite similar APIs, the semantics of various STM implementations often differ significantly (see e.g., (LR07; DS07)). This is a result of many design decisions that have to be considered, which make these implementations almost incomparable one another and to the language defined in this paper. Some design decisions are determined by the choice of a programming language. For example, STM for Haskell (PJGF96) is based on monads, STM for OCaml (RG05) is based on light-weight threads, STM for C (SATH$^+$06) requires machine code instrumentation while STM for Java (HF03) usually operates on bytecode. Other design decisions are the result of expected STM's functionality which may lead to different semantics—we may have different support for rollback, exception handling, I/O operations, and native operations. Contrary to our language, most of STMs forbids the use of I/O operations (native methods) with irrevocable effects inside trans-

actions. Sometimes this property is verified statically e.g., in STM for Haskell (HMPH05; PJGF96).

The semantics also depends on issues like: the concurrency control used (optimistic *vs.* pessimistic) which influences behaviour of transactional code with respect to non-transactional code, and the existence of a memory model (MPA05) for a particular programming language which influences behaviour of concurrent, non-synchronised code. An example isolation policy implemented by STMs is the *serializability* property, which corresponds to isolation defined in this paper. The property can be relaxed. For example, many database systems do not guarantee serializable transactions but provide relaxed isolation properties which are called *transaction isolation levels* (ALO00). Contrary to database systems, the designers of STM systems must take into account interaction of transactional code with the code outside the scope of transactions. This influences the semantics of atomicity and isolation. To address this problem, various authors differentiate between strong and weak atomicity (MBL06). Our language can support both strong and weak atomicity if, respectively, all data accesses are protected by verlocs, or some data accesses are left unprotected.

Several authors described pitfalls when replacing lock-based synchronization constructs by a particular model of STM (see e.g. (MBL06)). Transition from lock-based synchronisation to transactional synchronization cannot be performed automatically, just by substitution of synchronised blocks with `atomic`. It is a consequence of optimistic concurrency control (and weak atomicity) which can lead to deadlock in some situations. However, this specific problem does not occur if pessimistic concurrency control were used. In particular, our language allows standard locks and atomic transactions to be used together. Deadlocks are possible but they can only be caused by the standard locking principle, without anomalies described in (MBL06).

Another line of research is on type systems for specifying and verifying the atomicity of methods in multithreaded programs (see (FQ03) and follow-up papers). Our work has also similarities to work on lock inference, where a number of static analysis techniques is used to create locks for shared objects or memory locations in code to ensure atomicity or similar properties. In (HFP06) the authors present a method of inferring locks for atomic sections using type-based analysis, points-to analysis, and label flow analysis (PFH06) to determine which memory locations are shared and a domination relationship between shared locations to establish which locations may share a lock. In Autolocker (MZGB06) pessimistic atomic sections are converted into lock-guarded critical sections by analyzing dependencies among annotated locks based on a computation history derived from a transformation of the code using a type system. In (CGE08) backward data flow analysis is employed to transform the CFG into a path graph which then serves to derive locks.

There is no commonly accepted formal definition of semantics and abstract machines for STMs, although some proposals exist (e.g. (JV04; VJWH04), see also the fomalization of transactional systems (CR90)). Berger and Honda (BH00) have used a variant of π-calculus (MPW92) to formalize the operational semantics of the standard two-phase commitment protocol for distributed transactions. This work however does not address local concurrency control (on a machine) and the isolation property. In (BMM05), the authors formalize the mechanism of transaction compensations, which can be regared as

an alternative to rollback in atomic transactions. *Transaction compensations* are implicit or programmable procedures that can undo the effects of a transaction that fails to complete. Yu (Yu93) defines an analytical model of various concurrency control schemes used in transactional processing.

The closest work to us is the work of Jagannathan and Vitek (JV04; VJWH04) who proposed a calculi-based model of atomic transactions. They have formalized the optimistic concurrency control and two-phase locking strategies for software-based atomic transactions. Similarly to our approach, their formalization of the isolation property refers to the order (or scheduling) of concurrent actions. However, the soundness result rests upon an abstract notion of permutable actions, while our soundness result and proofs make explicit data accesses and transaction noninterference. This degree of detail allowed us to formally encode an example, version-based concurrency control algorithm.

Research on transaction management began appearing in the early to mid 1970s. Quite a large number of concurrency control algorithms have been proposed for use in centralised and distributed database systems. Database systems use concurrency control to avoid interference between concurrent transactions, which can lead to an inconsistent database. Isolation is used as the definition of correctness for concurrency control algorithms in these systems. The algorithms generally fall into one of three basic classes: *locking* algorithms, *timestamp* algorithms, and *optimistic* (or certification) algorithms. A comprehensive study of example techniques with pointers to literature can be found in (BHG87). Concurrency control problems had been also treated in the context of operating systems beginning in the mid 1960s. Most textbooks on operating systems survey this work, see e.g. (SGG02; Tan01).

Our versioning algorithms have some resemblance with basic two-phase locking. However, instead of acquiring all locks needed (in the 1st phase) and releasing them (in the 2nd phase), tasks take and dynamically upgrade version numbers, which optimizes unnecessary blocking. The conflicting operations are ordered according to version numbers, which is similar to ordering *timestamps* in timestamp algorithms (BHG87; WV02). However, we associate versions ("timestamps") with services, not with transactions. Therefore all service calls are always made in the right order for the isolation property (the call requests with too high versions are simply delayed), unlike common timestamp algorithms for database atomic transactions, where if an operation has arrived too late (that is it arrives after the transaction scheduler has already output some conflicting operation), the transaction must abort and be rolled back. The "ultimate conservative" timestamp algorithms avoid aborting by scheduling all operations in timestamp order, however, they produce serial executions (except complex variants that use transaction classes) (BHG87).

Methods of *deadlock avoidance* in allocating resources (SGG02; Tan01) are also relevant to our work. The *banker's algorithm* (introduced by Dijkstra (Dij65)) considers each request by a process as it occurs, and assigns the requested resource to the process only if there is a guarantee that this will leave the system in a safe state, that is no deadlock can occur. Otherwise the process must wait until some other process releases enough resources. The *resource-allocation graph* (Hol72) algorithm makes allocation decisions using a directed graph that dynamically records claims, requests and allocations

of resources by processes. The request can be granted only if the graph's transformation does not result in a cycle. Resources must be claimed a priori in these algorithms.

In our case, a task must also know a priori all its resources (methods or protocols) before it can commence. However, the history of service calls by different tasks is always acyclic since versions impose a total order on call requests performed by different tasks. Since tasks are assumed to complete, old versions will be eventually upgraded. Therefore our versioning algorithms are deadlock-free. Moreover, the calls are assigned according to the order that is necessary to satisfy the isolation property, unlike the resource allocation algorithms, which do not deal with ordering of operations on resources.

6. Conclusion and Future Work

The calculus of atomic transactions defined in this paper lays foundation for an intermediate concurrent language aimed at multicore CPUs, which employs software-based atomic transactions relaying on pessimistic concurrency control. The transactions can perform arbitrary operations on shared data, including I/O operations and native methods with irrevocable effects, with the isolation (or atomicity) guarantee. Only operations guarded by a special construct are isolated, which allows isolation to be relaxed for operations that should not be isolated.

The language of atomic transactions is typed, with a type system able to verify if an example versioning algorithm (BVA) obtains correct input data. This feature provides guarantees that the runtime execution of atomic transactions satisfies isolation. The operational semantics of the language has enabled formal proofs of language safety, including the proof of dynamic correctness of the BVA versioning algorithm. For efficiency, the type system could be easily extended to add distinction between read-only and read-write locking. It may be also worthwhile to investigate algorithms for inferring the typing annotations.

For clarity, we have presented somewhat idealised concurrency control algorithms. For instance, if a thread of some transaction is preempted while holding a verlock then no other thread can access the verlock; this can be solved in any practical implementation using some detection mechanism. In the future, we would like to find more robust ways of implementing `atomic`. Independently, we develop a mechanism of distributed atomic transactions, which is based on the ideas developed in this paper.

References

Atul Adya, Barbara Liskov, and Patrick E. O'Neil. Generalized isolation level definitions. In *Proceedings of ICDE '00: Conference on Data Engineering*, 2000.

Atomic RMI. `http://www.it-soa.eu/atomicrmi`, 2011.

Martin Berger and Kohei Honda. The two-phase commitment protocol in an extended pi-calculus. *Electronic Notes in Theoretical Computer Science*, 39(1), 2000.

Philip Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

Andrew Birrell. An introduction to programming with c# threads. Technical Report TR-2005-68, Microsoft Research Technical Report, May 2005.

Roberto Bruni, Hernán Melgratti, and Ugo Montanari. Theoretical foundations for compensations in flow composition languages. In *Proceedings of POPL '05: the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2005.

David Cunningham, Khilan Gudka, and Susan Eisenbach. Keep off the grass: Locking the right path for atomicity. In *Proceedings of CC '08: the 17th International Conference on Compiler Construction*, LNCS 4959, April 2008.

Panos K. Chrysanthis and Krithi Ramamritham. ACTA: A framework for specifying and reasoning about transaction structure and behavior. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1990.

Edsger W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965.

Dave Dice and Nir Shavit. Understanding tradeoffs in software transactional memory. In *Proceedings of CGO '07: the International Symposium on Code Generation and Optimization*, pages 21–33, 2007.

Cormac Flanagan and Martin Abadi. Types for safe locking. In *Proceedings of ESOP '99: the 8th European Symposium on Programming*, volume 1576 of *LNCS*. Springer, March 1999.

Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proceedings of PLDI '03: Conference on Programming Language Design and Implementation*, June 2003.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison Wesley, 2000.

Tim Harris. Exceptions and side-effects in atomic blocks. *Science of Computer Programming*, 58(3):325–343, 2005.

Timothy Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of OOPSLA '03: the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2003.

Michael Hicks, Jeffrey S. Foster, and Polyvios Prattikakis. Lock inference for atomic sections. In *Proceedings of TRANSACT '06: the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

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 (TOPLAS)*, 16(6):1719–1736, November 1994.

Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *Proceedings of OOPSLA '06: the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 253–262, October 2006.

Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of PODC '03: the 22nd ACM Symposium on Principles of Distributed Computing*, pages 92–101, July 2003.

Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of PPoPP '05: the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2005.

C. A. R. Hoare. Towards a theory of parallel programming. In *Operating Systems Techniques*, volume 9 of *A.P.I.C. Studies in Data Processing*, pages 61–71. Academic Press, 1972.

Richard C. Holt. Some deadlock properties of computer systems. *ACM Computing Surveys*, 4(3):179–196, 1972.

Suresh Jagannathan and Jan Vitek. Optimistic concurrency semantics for transactions in coordination languages. In *Proceedings of COORDINATION '04: the 6th International Conference on Coordination Models and Languages*, volume 2949 of *LNCS*. Springer, February 2004.

Jim Larus and Ravi Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, 2007.

Milo Martin, Colin Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5(2):17, 2006.

Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Proceedings of POPL '05: the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2005.

R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I and II. *Information and Computation*, 100(1):1–77, 1992.

Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. In *Proceedings of POPL '06: the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 346–358, 2006.

Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design and implementation of transactional constructs for C/C++. In *Proceedings of OOPSLA '08: the 23rd ACM SIGPLAN Conference on Object-oriented Programming, Systems Languages and Applications*, October 2008.

Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Existential label flow inference via CFL reachability. In *Proceedings of SAS 2006: the 13th International Symposium on the Static Analysis*, LNCS 4134, pages 88–106, 2006.

Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Proceedings of POPL '96: the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, 1996.

Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

Michael F. Ringenburg and Dan Grossman. AtomCaml: first-class atomicity via rollback. In *Proceedings of ICFP '05: the 10th ACM SIGPLAN International Conference on Functional Programming*, September 2005.

Java RMI. `http://java.sun.com/products/jdk/rmi/`.

Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of PPoPP '06: the eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–197, New York, NY, USA, 2006. ACM.

Avi Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts, Sixth Edition*. John Wiley & Sons, Inc, 2002.

William N. Scherer, III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Proc. of PODC '05*, pages 240–248, 2005.

Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of PODCS '95: the 14th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1995.

Andrew S. Tanenbaum. *Modern Operating Systems, Second Edition*. Prentice Hall, Englewood Cliff, NJ, 2001.

Jan Vitek, Suresh Jagannathan, Adam Welc, and Antony L. Hosking. A semantic framework for designer transactions. In *Proceedings of ESOP '04: the 13th European Symposium on Programming*, volume 2986 of *LNCS*. Springer, March/April 2004.

Raja Vallée-Rai and Laurie J. Hendren. Jimple: Simplifying Java bytecode for analyses and transformations. Technical Report 1998-4, McGill University, July 1998.

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

Jeannette M. Wing, Manuel Faehndrich, J. Gregory Morrisett, and Scott Nettles. Extensions to Standard ML to support transactions. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, June 1992.

Paweł T. Wojciechowski. Isolation-only transactions by typing and versioning. In *Proceedings of PPDP '05: the 7th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming*, July 2005.

Paweł T. Wojciechowski. *Language Design for Atomicity, Declarative Synchronization, and Dynamic Update in Communicating Systems*. The Poznań University of Technology Press, 2007. Habilitation thesis.

Paweł T. Wojciechowski, Olivier Rütti, and André Schiper. SAMOA: A framework for a synchronisation-augmented microprotocol approach. In *Proceedings of IPDPS '04: the 18th IEEE International Parallel and Distributed Processing Symposium*, April 2004.

Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems*. Morgan Kaufmann, 2002.

Philip S. Yu. Modeling and analysis of transaction processing systems. In *Performance Evaluation of Computer and Communication Systems*, LNCS 729. Springer-Verlag, 1993.

Appendix A.

A.1. *Type Soundness*

Reduction of a program may either continue forever, or may reach a final state, where no further evaluation is possible. Such a final state represents either an answer or a type error. Since programs expressed in our language are not guaranteed to be deadlock-free (unless all transactions are single-threaded and re-entrant locking is allowed), we also admit a deadlocked state to be an (acceptable) answer. Thus, proving type soundness means that well-typed programs yield only well-typed answers.

Our proof of type soundness rests upon the notion of type preservation (also known as subject reduction). The type preservation property states that reductions preserve the type of expressions.

Type preservation by itself is not sufficient for type soundness. In addition, we must prove that programs containing type errors are not typable. We call such expressions with type errors *faulty expressions* and prove that faulty expressions cannot be typed.

A.1.1. *Type safety* The statement of the main type preservation lemma must take stores and store typings into account. For this we need to relate stores with assumptions about the types of the values in the stores. Below we define what it means for a store π, σ to be well typed. (For clarity, we omit permissions p from the context and global gv and local lv counters from states when possible.)

Definition 6 *A store π, σ is said to be* well typed *with respect to a store typing Σ and a typing context Γ, written $\Sigma \mid \Gamma; a \vdash \pi, \sigma$, if $dom(\pi, \sigma) = dom(\Sigma)$ and $\Sigma \mid \Gamma; a \vdash \mu(l) : \Sigma(l)$ for every store $\mu \in \{\pi, \sigma\}$ and every $l \in dom(\mu)$.*

Intuitively, a store π, σ is consistent with a store typing Σ if every value in the store has the type predicted by the store typing.

By canonical forms (Lemma 31 in Section A.1.2), each *location value* $l \in dom(\pi, \sigma)$ can be either a verlock location, or a reference location, depending on a concrete type. For simplicity, we often refer to π, σ as the store, meaning individual stores, i.e. either π or σ, depending on a given value and type. If a location value l is a verlock location then it is kept in a verlock store π; if the value is a reference location then it is kept in a reference store σ.

Type preservation for our language states that the reductions defined in Figures 4, 5 and 8 preserve type:

Theorem 7 (Type Preservation) *If $\Sigma \mid \Gamma; a \vdash T : t$ and $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ and $\langle \pi, \sigma \mid T \rangle \longrightarrow \langle (\pi, \sigma)' \mid T' \rangle$, then for some $\Sigma' \supseteq \Sigma$, $\Sigma' \mid \Gamma; a \vdash T' : t$ and $\Sigma' \mid \Gamma; a \vdash (\pi, \sigma)'$.*

The type preservation theorem asserts that there is some store typing $\Sigma' \supseteq \Sigma$ (i.e., agreeing with Σ on the values of all the old locations) such that a new term T' is well typed with respect to Σ'. This new store typing Σ' is either Σ or it is exactly $(\Sigma, l : t_0)$, where l is a newly allocated location, i.e. the new element of $dom((\pi, \sigma)')$, and t_0 is the type of the initial value bound to l in the extended store $(\mu, l \mapsto v_0)$ for some $\mu \in \{\pi, \sigma\}$.

Proof. The proof is a straightforward induction on a derivation of $T : t$, using the lemmas below and the inversion property of the typing rules. The proof proceeds by case analysis according to the reduction $T \longrightarrow T'$.

Case $\langle \pi, \sigma \mid \lambda^{b,p} x : s.\, e\ v \rangle \longrightarrow \langle \pi, \sigma \mid e\{v/x\} \rangle$.

From $\Sigma \mid \Gamma; a \vdash \lambda^{b,p} x : s.\, e\ v : t$ we have $\Sigma \mid \Gamma; a \vdash v : s$ and $\Sigma \mid \Gamma; a \vdash \lambda^{b,p} x : s.\, e : s \to^{b,p} t$ and $b \subseteq a$ by (T-App). From the latter, $\Sigma \mid (\Gamma, x : s); b \vdash e : t$ follows by (T-Fun). Hence $\Sigma \mid \Gamma; b \vdash e\{v/x\} : t$ by substitution Lemma 27 and $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ from premise.

Case $\langle \pi, \sigma \mid \mathtt{ref}_m\, v \rangle \longrightarrow \langle \pi, (\sigma, r \mapsto v) \mid r \rangle$ if $r \notin dom(\sigma)$.

From $\Sigma \mid \Gamma; a \vdash \mathtt{ref}_m\, v : t$ where $t = \mathtt{Ref}_m\, t'$, we have

$$\Sigma \mid \Gamma; a \vdash v : t' \tag{8}$$

and $\Gamma \vdash m$ by (T-Ref), and $(\Sigma, r : t') \mid \Gamma; a \vdash v : t'$ by store typing Lemma 30, where r is a fresh reference cell location. Hence $(\Sigma, r : t') \mid \Gamma; a \vdash r : \mathtt{Ref}_m\, t'$ by (T-RefLoc), which is the first part of the needed result.

From the latter, since $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (premise) and $r : t' \notin \Sigma$ (immediate from the premise that π, σ is well-typed and the assumption that $r \notin dom(\sigma)$) hence $(\Sigma, r : t') \mid \Gamma; a \vdash \pi, (\sigma, r \mapsto v)$ by (8) and store extension (Lemma 29), which completes the second part of the needed result.

Case $\langle \pi, \sigma \mid !r \rangle \longrightarrow \langle \pi, \sigma \mid v \rangle$ if $\sigma(r) = v$.

From $\Sigma \mid \Gamma; a \vdash !r : t$, we have $\Sigma \mid \Gamma; a \vdash r : \mathtt{Ref}_m\, t$ by (T-Deref). From the latter, we have $\Sigma(r) = t$ and $\Sigma \mid \Gamma \vdash m$ by (T-RefLoc), and so $\Sigma \mid \Gamma; a \vdash \sigma(r) : \Sigma(r)$ by premise that the store π, σ is well typed and Definition 6. Hence $\Sigma \mid \Gamma; a \vdash v : t$ (immediate from the assumption that $\sigma(r) = v$) and $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ from premise.

Case $\langle \pi, \sigma \mid r := v \rangle \longrightarrow \langle \pi, \sigma[r \mapsto v] \mid () \rangle$.

From $\Sigma \mid \Gamma; a \vdash r := v : t$ where $t = \mathtt{Unit}$, and $\Sigma \mid \Gamma; a \vdash () : \mathtt{Unit}$ by (T-Unit), we have immediately the first part of the needed result. From $\Sigma \mid \Gamma; a \vdash r := v : \mathtt{Unit}$, we have $\Sigma \mid \Gamma; a \vdash r : \mathtt{Ref}_m\, t'$ and

$$\Sigma \mid \Gamma; a \vdash v : t' \tag{9}$$

by (T-Assign). From the former, we have $\Sigma(r) = t'$ and $\Sigma \mid \Gamma \vdash m$ by (T-RefLoc), hence

$\Sigma \mid \Gamma; a \vdash \pi, \sigma[r \mapsto v]$ by (9), premise that the store π, σ is well typed, and the store update Lemma 28, which completes the second part of the needed result.

Case $\langle \pi, \sigma \mid \mathcal{E}[\,\texttt{fork}\ e\,]\rangle \longrightarrow \langle \pi, \sigma \mid \mathcal{E}[\,()\,], e\rangle$.

From $\Sigma \mid \Gamma; a \vdash \mathcal{E}[\,\texttt{fork}\ e\,] : t$ we have

$$\Sigma' \mid \Gamma'; a \ \vdash\ e : \texttt{Unit} \tag{10}$$

$$\Sigma' \mid \Gamma'; a \ \vdash\ \texttt{fork}\ e : \texttt{Unit} \tag{11}$$

for some Σ' and Γ' by (T-Fork). From $\Sigma \mid \Gamma; a \vdash \mathcal{E}[\,\texttt{fork}\ e\,] : t$ and (11) we have $\Sigma \mid \Gamma; a \vdash \mathcal{E}[\,()\,] : t$ by (T-Unit) and replacement Lemma 26. From the latter and (10) we have $\Sigma \mid \Gamma; a \vdash \mathcal{E}[\,()\,], e : t$ by (T-Unit) and (T-Thread), which together with $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (premise), completes both parts of the needed result.

Case $\langle \pi, \sigma \mid f_i, f'_j\rangle \longrightarrow \langle \pi, \sigma \mid f_i\rangle$ if $i < j$.

From $\Sigma \mid \Gamma; a \vdash f_i, f'_j : t$ and $i < j$ we have immediately $\Sigma \mid \Gamma; a \vdash f_i : t$ and $\Sigma \mid \Gamma; a' \vdash f'_j : t'$ for some a' and t' by (T-Thread). The former derivative and $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (premise) complete both parts of the needed result.

Case $\langle \pi, \sigma \mid \mathcal{E}[\,\texttt{atomic}\ \bar{l}\ e\,]\rangle \longrightarrow \langle \pi, \sigma \mid \mathcal{E}[\,()\,], \texttt{transact}\ pv\ e\rangle$.

From $\Sigma \mid \Gamma; a \vdash \mathcal{E}[\,\texttt{atomic}\ \bar{l}\ e\,] : t$, by (T-Isol) we have $\Sigma' \mid \Gamma'; a \vdash l_i : o_{li}$ for all $i = 1..|\bar{l}|$, and $\Sigma' \mid \Gamma'; \{o_{l1}\} \cup ... \cup \{o_{l|\bar{l}|}\} \vdash e : t'$ for some t', and $\Sigma' \mid \Gamma'; a \vdash \texttt{atomic}\ \bar{l}\ e : \texttt{Unit}$ for some Σ' and Γ'. Hence $\Sigma \mid \Gamma; a \vdash \mathcal{E}[\,()\,] : t$ by (T-Unit) and replacement Lemma 26. Since $\Sigma' \mid \Gamma'; a \vdash \texttt{transact}\ pv\ e : \texttt{Unit}$ by (T-Transact), hence $\langle \pi, \sigma \mid \mathcal{E}[\,()\,], \texttt{transact}\ pv\ e\rangle : t$ by (T-Unit) and (T-Thread), which together with $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (premise), completes both parts of the needed result.

Case $\langle \pi, \sigma \mid \texttt{transact}\ pv\ v\rangle \longrightarrow \langle \pi, \sigma \mid ()\rangle$.

From $\Sigma \mid \Gamma; a \vdash \texttt{transact}\ pv\ v : t$ we have $t = \texttt{Unit}$ by (T-Transact), and $\Sigma \mid \Gamma; a \vdash () : \texttt{Unit}$ by (T-Unit), which together with $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (premise), completes both parts of the needed result.

Case $\langle \pi, \sigma \mid \texttt{newlock}\ x{:}m\ \texttt{in}\ e\rangle \longrightarrow \langle (\pi, l \mapsto 0), \sigma \mid e\{l/x\}\{o_l/m\}\rangle$ if $l \notin dom(\pi)$.

From $\Sigma \mid \Gamma; a \vdash \texttt{newlock}\ x : m\ \texttt{in}\ e : t$ and $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (premise), we have $\Sigma \mid (\Gamma, m :: \texttt{Lock}, x : m); a \vdash e : t$ and $\Sigma \mid \Gamma \vdash a$ and $\Sigma \mid \Gamma \vdash t$ by (T-Lock), and hence

$$(\Sigma, l : \{0,1\}, o_l :: \texttt{Lock}) \mid (\Gamma, m :: \texttt{Lock}, x : m); a \vdash e : t \tag{12}$$

by store typing (Lemma 30). Since $(\Sigma, l : \{0,1\}, o_l :: \texttt{Lock}) \mid \Gamma; a \vdash l : o_l$ by (T-LockLoc), hence $(\Sigma, l : \{0,1\}, o_l :: \texttt{Lock}) \mid \Gamma; a \vdash e\{l/x\}\{o_l/m\} : t$ by (12), substitution (Lemma 27) and the definition of a singleton verlock type, which is the first part of the needed result.

From the latter, since $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (premise), $l : \{0,1\} \notin \Sigma$ (immediate from the premise that π, σ is well-typed and the assumption that $l \notin \mathit{dom}(\pi)$), and $\Sigma \mid \Gamma; a \vdash 0 : \{0,1\}$ hence $(\Sigma, l : \{0,1\}, o_l :: \texttt{Lock}) \mid \Gamma; a \vdash (\pi, l \mapsto 0), \sigma$ by store extension (Lemma 29).

Case $\langle \pi, \sigma \mid \texttt{sync}\ l\ e \rangle \longrightarrow \langle \pi[l \mapsto 1], \sigma \mid \texttt{insync}\ l\, e \rangle$ if $\pi(l) = 0$.

From $\Sigma \mid \Gamma; a \vdash \texttt{sync}\ l\ e : t$, we have

$$\Sigma \mid \Gamma; a \vdash l : o_l \quad o_l \in a \tag{13}$$

$$\Sigma \mid \Gamma; a \vdash e : t \tag{14}$$

by (T-Sync). From (13) and $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (premise), we have

$$\Sigma(l) = \{0,1\} \tag{15}$$

and $\Sigma(o_l) = \texttt{Lock}$ by (T-LockLoc). From (13) and (14) and $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (premise), we have $\Sigma \mid \Gamma; a \vdash \texttt{insync}\ l\, e : t$ by (T-InSync), which completes the first part of the needed result.

From $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (premise) and (15) and $\Sigma \mid \Gamma; a \vdash 1 : \{0,1\}$, we have $\Sigma \mid \Gamma; a \vdash \pi[l \mapsto 1], \sigma$ by the store update Lemma 28, which completes the second part of the needed result.

Case $\langle \pi, \sigma \mid \texttt{insync}\ l\, v \rangle \longrightarrow \langle \pi[l \mapsto 0], \sigma \mid v \rangle$ if $\pi(l) = 1$.

From $\Sigma \mid \Gamma; a \vdash \texttt{insync}\ l\, v : t$, we have

$$\Sigma \mid \Gamma; a \vdash l : o_l \tag{16}$$

$$\Sigma \mid \Gamma; a \vdash v : t \tag{17}$$

and $o_l \in a$ by (T-InSync), which completes the first part of the needed result. From $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (premise) and (16), we have $\Sigma(l) = \{0,1\}$ and $\Sigma(o_l) = \texttt{Lock}$ by (T-LockLoc). From the latter and $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (premise) and $\Sigma \mid \Gamma; a \vdash 0 : \{0,1\}$, we have $\Sigma \mid \Gamma; a \vdash \pi[l \mapsto 0], \sigma$ by the store update Lemma 28, which completes the second part of the needed result. □

This completes the main part of the proof. It remains to establish several technical lemmas.

Some obvious facts about deductions that we use:

—if $\Sigma \mid \Gamma \vdash \mathcal{E}[\,e\,] : t$ then there exist Σ', Γ' and t' such that $\Sigma' \mid \Gamma' \vdash e : t'$;

—if there are no Σ', Γ' and t' such that $\Sigma' \mid \Gamma' \vdash e : t'$, then there are no Σ, Γ, and t such that $\Sigma \mid \Gamma \vdash \mathcal{E}[\,e\,] : t$.

These follow from the facts that (1) there is exactly one inference rule for each expression form e, and (2) each inference rule requires a proof for each subexpression of the expression in its conclusion.

The first lemma states that we may permute the elements of a context, as convenient, without changing the set of typing elements that can be derived from under it.

Lemma 24 (Permutation) *If* $\Sigma \mid \Gamma; a \vdash T : t$ *and* Δ *is a permutation of* Γ*, then* $\Sigma \mid \Delta; a \vdash T : t$*. Moreover, the latter derivation has the same depth as the former.*

Proof. Straightforward induction on typing derivations. □

The following lemma states that extra variables in the typing environment Γ of a judgment $\Gamma \vdash e : t$ that are not free in the expression e may be ignored.

Lemma 25 (Weakening) *If* $\Gamma(x) = \Gamma'(x)$ *for all* $x \in fv(e)$ *then* $\Sigma \mid \Gamma; a \vdash e : t$ *iff* $\Sigma \mid \Gamma'; a \vdash e : t$*.*

Proof. Straightforward induction on typing derivations. □

A key lemma that we use in the proof of type preservation is the replacement lemma. It allows the replacement of one of the subexpressions of a typable expression with another subexpression of the same type, without disturbing the type of the overall expression.

Lemma 26 (Replacement) *If:*

1 $\mathcal{D}$ *is a deduction concluding* $\Sigma \mid \Gamma; a \vdash \mathcal{E}[\, e_1 \,] : t$*,*

2 $\mathcal{D}'$ *is a subdeduction of* $\mathcal{D}$ *concluding* $\Sigma' \mid \Gamma'; a' \vdash e_1 : t'$*,*

3 $\mathcal{D}'$ *occurs in* $\mathcal{D}$ *in the position corresponding to the hole* ($\mathcal{E}$) *in* $\mathcal{E}[\,]$*, and*

4 $\Sigma' \mid \Gamma'; a' \vdash e_2 : t'$

then $\Sigma \mid \Gamma; a \vdash \mathcal{E}[\, e_2 \,] : t$*.*

Proof. See (WF94) (for a language with no stores and store typing; the proof is also valid for our language). □

The substitution lemma is the key to showing type preservation for reductions involving substitution.

Lemma 27 (Substitution) *If* $\Sigma \mid (\Gamma, x : t); a \vdash e : t'$ *and* $\Sigma \mid \Gamma; a \vdash v : t$*, then* $\Sigma \mid \Gamma; a \vdash e\{v/x\} : t'$*.*

Proof. We proceed by induction on a derivation of the statement $(\Gamma, x : t) \vdash e : t'$, and case analysis on the final typing rule used in the proof. (For clarity, we remove store typing Σ, allocation and permission whenever possible.)

Case $e = ()$.

If so then $\Gamma \vdash () : t'$ and $t' = \texttt{Unit}$ by (T-Unit). Then $\Gamma \vdash ()\{v/x\} : t'$ since $()\{v/x\} = ()$ (the same would be for any other constants).

Case $e = x'$.

There are two sub-cases to consider, depending on whether x' is x or another variable.

(1) If $x' \neq x$, then $x' : t' \in \Gamma$ by (T-Var), and $\Gamma \vdash x' : t'$ again by (T-Var). Then $\Gamma \vdash x'\{v/x\} : t'$ since $x'\{v/x\} = x'$.

(2) If $x' = x$, then $x : t' \in \Gamma$ by (T-Var), and $\Gamma \vdash x : t'$ again by (T-Var). Since $x\{v/x\} = v$, hence $\Gamma \vdash x\{v/x\} : t'$.

Case $e = \lambda^{b,p} x' : t_1 . e_1$.

By (T-Fun), it follows from the assumption $(\Gamma, x : t) \vdash \lambda^{b,p} x' : t_1 . e_1 : t'$ that $t' = t_1 \to^{b,p} t_2$ and $(\Gamma, x : t, x' : t_1) \vdash e_1 : t_2$. Using permutation on the given subderivation, we obtain $(\Gamma, x' : t_1, x : t) \vdash e_1 : t_2$. Using weakening (Lemma 25) on the other given derivation $(\Gamma \vdash v : t)$, we obtain $(\Gamma, x' : t_1) \vdash v : t$.

Now, by the inductive hypothesis, $(\Gamma, x' : t_1) \vdash e_1\{v/x\} : t_2$. By (T-Fun), we have $\Gamma \vdash \lambda^{b,p} x' : t_1 . e_1\{v/x\} : t_1 \to^{b,p} t_2$. But this is precisely the needed result, since, by the definition of substitution, $\Gamma \vdash (\lambda^{b,p} x' : t_1 . e_1)\{v/x\} : t_1 \to^{b,p} t_2$.

Case $e = e_1\ e_2$.

From $(\Gamma, x : t); a \vdash e_1\ e_2 : t'$ by the first premise of (T-App), we have $(\Gamma, x : t); a \vdash e_1 : t_1 \to^{b,p} t'$ for some t_1 and $b \subseteq a$, and

$$\Gamma; a \vdash e_1\{v/x\} : t_1 \to^{b,p} t' \tag{18}$$

by induction hypothesis. By the second premise of (T-App), we have $(\Gamma, x : t); a \vdash e_2 : t_1$, and

$$\Gamma; a \vdash e_2\{v/x\} : t_1 \tag{19}$$

by induction hypothesis.

Then by (T-App) with (18) and (19) $\Gamma; a \vdash e_1\{v/x\}\ e_2\{v/x\} : t'$. But this is precisely the needed result, since, by the definition of substitution $\Gamma; a \vdash (e_1\ e_2)\{v/x\} : t'$.

Case $e = \mathtt{ref}_m\, e : \mathtt{Ref}_m\, t_1$.

By (T-Ref), it follows from the assumption $(\Gamma, x : t) \vdash \mathtt{ref}_m\, e : t'$ that $t' = \mathtt{Ref}_m\, t_1$, and $(\Gamma, x : t) \vdash e : t_1$ and $\Gamma \vdash m$.

Now, by the induction hypothesis, $\Gamma \vdash e\{v/x\} : t_1$. By (T-Ref), we have $\Gamma \vdash \mathtt{ref}_m\, e\{v/x\} : \mathtt{Ref}_m\, t_1$. But this is precisely the needed result, since, by the definition of substitution $\Gamma \vdash (\mathtt{ref}_m\, e)\{v/x\} : \mathtt{Ref}_m\, t_1$.

Case $e = !e$.

By (T-Deref), it follows from the assumption $(\Gamma, x : t); a \vdash !e : t'$ that $(\Gamma, x : t); a \vdash e : \mathtt{Ref}_m\, t'$ for some $m \in a$.

Now, by the induction hypothesis, $\Gamma; a \vdash e\{v/x\} : \texttt{Ref}_m\, t'$. By (T-Deref), we have $\Gamma; a \vdash !e\{v/x\} : t'$. But this is precisely the needed result, since, by the definition of substitution $\Gamma; a \vdash (!e)\{v/x\} : \texttt{Ref}_m\, t'$.

Case $e = e_1 := e_2$.

From $(\Gamma, x : t); a \vdash e_1 := e_2 : t'$, where $t' = \texttt{Unit}$, by the first premise of (T-Assign), we have $(\Gamma, x : t); a \vdash e_1 : \texttt{Ref}_m\, t_1$ for some t_1, and

$$\Gamma; a \vdash e_1\{v/x\} : \texttt{Ref}_m\, t_1 \tag{20}$$

by induction hypothesis. By the second premise of (T-Assign), we have $(\Gamma, x : t); a \vdash e_2 : t_1$ and $m \in a$, and

$$\Gamma; a \vdash e_2\{v/x\} : t_1 \tag{21}$$

by induction hypothesis.

Then by (T-Assign) with (20) and (21) $\Gamma; a \vdash e_1\{v/x\} := e_2\{v/x\} : \texttt{Unit}$. But this is precisely the needed result, since, by the definition of substitution $\Gamma; a \vdash (e_1 := e_2)\{v/x\} : \texttt{Unit}$.

Case $e = \texttt{newlock}\ x' : m\ \texttt{in}\ e'$.

By (T-Lock), it follows from the assumption $(\Gamma, x : t); a \vdash \texttt{newlock}\ x' : m\ \texttt{in}\ e' : t'$ that $(\Gamma, x : t, m :: \texttt{Lock}, x' : m); a \vdash e' : t'$ and $\Gamma \vdash a$ and $\Gamma \vdash t'$.

Now, by the induction hypothesis, $(\Gamma, m :: \texttt{Lock}, x' : m); a \vdash e'\{v/x\} : t'$. By (T-Lock), we have $\Gamma; a \vdash \texttt{newlock}\ x' : m\ \texttt{in}\ e'\{v/x\} : t'$. But this is precisely the needed result, since, by the definition of substitution, $\Gamma; a \vdash (\texttt{newlock}\ x' : m\ \texttt{in}\ e')\{v/x\} : t'$.

Case $e = \texttt{sync}\ e_1\ e_2$.

From $(\Gamma, x : t); a; p \vdash \texttt{sync}\ e_1\ e_2 : t'$ by the first premise of (T-Sync), we have $(\Gamma, x : t); a; p \vdash e_1 : m$ and

$$m \in a \ . \tag{22}$$

By induction hypothesis

$$\Gamma; a \vdash e_1\{v/x\} : m \ . \tag{23}$$

By the second premise of (T-Sync), we have $(\Gamma, x : t); a; p \cup \{m\} \vdash e_2 : t'$. By induction hypothesis

$$\Gamma; a; p \cup \{m\} \vdash e_2\{v/x\} : t' \ . \tag{24}$$

Then by (T-Sync) with (22), (23) and (24) we have $\Gamma; a; p \vdash \texttt{sync}\ e_1\{v/x\}\ e_2\{v/x\} : t'$. But this is precisely the needed result, since, by the definition of substitution $\Gamma; a; p \vdash (\texttt{sync}\ e_1\ e_2)\{v/x\} : t'$.

Case $e = \texttt{insync}\ e\, f$.

By (T-InSync), it follows from the assumption $(\Gamma, x : t); a; p \vdash \texttt{insync}\ e\, f : t'$ that $(\Gamma, x : t); a; p \vdash e : m$ and $(\Gamma, x : t); a; p \vdash f : t'$ and $m \in a$, $m \in p$.

Now, by induction hypothesis, $\Gamma; a; p \vdash e\{v/x\} : m$ and $\Gamma; a; p \vdash f\{v/x\} : t'$. By (T-InSync), we have $\Gamma; a; p \vdash \texttt{insync}\ e\{v/x\}\, f\{v/x\} : t'$. But this is precisely the needed result, since, by the definition of substitution $\Gamma; a; p \vdash (\texttt{insync}\ e\, f)\{v/x\} : t'$.

Case $e = \texttt{fork}\ e$.

By (T-Fork), it follows from the assumption $(\Gamma, x : t) \vdash \texttt{fork}\ e : t'$ that $(\Gamma, x : t) \vdash e : t'$ and $t' = \texttt{Unit}$.

Now, by the induction hypothesis, $\Gamma \vdash e\{v/x\} : t'$. By (T-Fork), we have $\Gamma \vdash \texttt{fork}\ e\{v/x\} : t'$. But this is precisely the needed result, since, by the definition of substitution $\Gamma \vdash (\texttt{fork}\ e)\{v/x\} : t'$.

Case $e = \texttt{atomic}\ e_1, ..., e_n\ e_0$.

From $(\Gamma, x : t); a; p \vdash \texttt{atomic}\ e_1, ..., e_n\ e_0 : t'$ and $t' = \texttt{Unit}$, by the first premise of (T-Isol), we have $(\Gamma, x : t); a; p \vdash e_i : m_i$ for all $i = 1..n$, and

$$\Gamma; a; p \vdash e_i\{v/x\} : m_i \quad \text{for all } \ i = 1..n \tag{25}$$

by induction hypothesis. By the second premise of (T-Isol), we have $(\Gamma, x : t); \{m_1\} \cup ... \cup \{m_n\}; p \vdash e_0 : t_0$ for some t_0, and

$$\Gamma; \{m_1\} \cup ... \cup \{m_n\}; \emptyset \vdash e_0\{v/x\} : t_0 \tag{26}$$

by induction hypothesis.

Then by (T-Isol) with (25) and (26) we have $\Gamma; a; p \vdash \texttt{atomic}\ e_1\{v/x\}, ..., e_n\{v/x\}$ $e_0\{v/x\}$: $\texttt{Unit}$. But this is precisely the needed result, since, by the definition of substitution $\Gamma; a; p \vdash (\texttt{atomic}\ e_1, ..., e_n\ e_0)\{v/x\} : \texttt{Unit}$.

Case $e = f_i, f'_j$ and $i < j$.

By (T-Thread), it follows from the assumption $(\Gamma, x : t) \vdash f_i, f'_j : t'$ and $i < j$, that $(\Gamma, x : t) \vdash f_i : t'$ and $(\Gamma, x : t) \vdash f'_j : t''$ for some t''.

Now, by the induction hypothesis, $\Gamma \vdash f_i\{v/x\} : t'$ and $\Gamma \vdash f'_j\{v/x\} : t''$. By (T-Thread) and $i < j$, we have $\Gamma \vdash f_i\{v/x\}, f'_j\{v/x\} : t'$. But this is precisely the needed result, since, by the definition of substitution $\Gamma \vdash (f_i, f'_j)\{v/x\} : t'$.

Case $e = \texttt{transact}\ pv\ f$.

By (T-Transact), it follows from the assumption $(\Gamma, x : t); a \vdash \texttt{transact}\ pv\ f : t'$ where $t' = \texttt{Unit}$, that $a = \{o_{l_1}, ..., o_{l_n}\}$ and $(\Gamma, x : t); a \vdash l_i : o_{l_i}$ and $(\Gamma, x : t); a \vdash pv(l_i) : \textbf{Nat}$

for all $i = 1..n$, and

$$\Gamma; a \vdash pv(l_i)\{v/x\} : \mathbf{Nat} \quad \text{for all } \; i = 1..n \tag{27}$$

by induction hypothesis. By the last premise of (T-Transact), we have $(\Gamma, x : t); a \vdash f : t$ for some t, and

$$\Gamma; a \vdash f\{v/x\} : t \tag{28}$$

by induction hypothesis.

Then by (T-Transact) with (27, 28) $\Gamma; a \vdash \texttt{transact}\ pv\{v/x\}\ f\{v/x\} : \texttt{Unit}$. But this is precisely the needed result, since, by the definition of substitution $\Gamma; a \vdash (\texttt{transact}\ pv\ f)\{v/x\} : \texttt{Unit}$. □

The next lemma states that replacing the contents of a store with a new value of appropriate type does not change the overall type of the store.

The notation $(\pi, \sigma)[l \mapsto v]$ should be read as $\pi[l \mapsto v], \sigma$ if l is a verlock location, or $\pi, \sigma[l \mapsto v]$ if l is a reference cell location. See the canonical forms Lemma 31 in Section A.1.2 that states the possible shapes of values of various types.

Lemma 28 (Store Update) *If* $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ *and* $\Sigma(l) = t$ *and* $\Sigma \mid \Gamma; a \vdash v : t$ *then* $\Sigma \mid \Gamma; a \vdash (\pi, \sigma)[l \mapsto v]$.

Proof. Immediate from the definition of $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (see Definition 6). □

The next lemma states that extending the contents of a store with a new value of appropriate type is consistent with the store typing.

The notation $((\pi, \sigma), l \mapsto v)$ should be read as $(\pi, l \mapsto v), \sigma$ if l is a verlock location, or $\pi, (\sigma, l \mapsto v)$ if l is a reference cell location. See the canonical forms Lemma 31 in Section A.1.2 that states the possible shapes of values of various types.

Lemma 29 (Store Extension) *If* $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ *and* $l : t \notin \Sigma$ *and* $\Sigma \mid \Gamma; a \vdash v : t$ *then* $(\Sigma, l : t) \mid \Gamma; a \vdash ((\pi, \sigma), l \mapsto v)$.

Proof. Immediate from the definition of $\Sigma \mid \Gamma; a \vdash \pi, \sigma$. □

Finally, we need a kind of weakening lemma for stores, stating that, if a store is extended with a new location then the extended store still allows us to assign types to all the same terms as the original.

Lemma 30 (Store Typing)
If $\Sigma \mid \Gamma; a \vdash e : t$ *and* $\Sigma' \supseteq \Sigma$, *then* $\Sigma' \mid \Gamma; a \vdash e : t$.

Proof. Easy by induction. □

A corollary of Type Preservation (Theorem 7) is that reduction steps preserve type.

Corollary 2 (Type Preservation) *If* $\Sigma \mid \Gamma; a \vdash T : t$ *and* $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ *and* $\langle \pi, \sigma \mid T \rangle \longrightarrow^* \langle (\pi, \sigma)' \mid T' \rangle$, *then for some* $\Sigma' \supseteq \Sigma$, $\Sigma' \mid \Gamma; a \vdash T' : t$ *and* $\Sigma' \mid \Gamma; a \vdash (\pi, \sigma)'$.

Proof. If $\langle \pi, \sigma \mid T \rangle \longrightarrow \langle (\pi, \sigma)' \mid T' \rangle$, then $T = \mathcal{E}[\, e_1 \,]$ and $T' = \mathcal{E}[\, e_2 \,]$, and $\langle \pi, \sigma \mid e_1 \rangle \longrightarrow \langle (\pi, \sigma)' \mid e_2 \rangle$ and $\Sigma' \mid \Gamma; a \vdash (\pi, \sigma)'$, for some $\Sigma' \supseteq \Sigma$, so $\Sigma' \mid \Gamma; a \vdash T' : t$ by the replacement Lemma 26. Then the result follows by induction on the length of the reduction sequence $\langle \pi, \sigma \mid T \rangle \longrightarrow^* \langle (\pi, \sigma)' \mid T' \rangle$. □

A.1.2. *Evaluation progress* Subject reduction ensures that if we start with a typable expression, then we cannot reach an untypable expression through any sequence of reductions. This by itself, however, does not yield type soundness.

Below, we prove that evaluation of a typable expression cannot get *stuck*, i.e. either the expression is a value or there is some reduction defined. However, we do allow reduction to be suspended indefinitely since our language is not deadlock-free. This is acceptable since we define and guarantee isolation, respectively isolation-up-to, only for programs that either terminate, or reach some result state (see Theorem 3 and Lemma 6).

A canonical forms lemma states the possible shapes of values of various types.

Lemma 31 (Canonical Forms)
1) If v is a value of type `Unit`*, then v is* $()$*.*
2) If v is a value of type $t \to^{a,p} s$, then $v = \lambda^{a,p} x : t.\, e$.
3) If v is a value of type m, then v is a verlock location.
5) If v is a value of type $\texttt{Ref}_m\, t$*, then v is a reference cell location (or reference location, in short) of a reference cell storing values of type t.*

Proof. Straightforward from the grammar in Figure 3 and the extended grammar in Figure 4. □

We state progress only for closed expressions, i.e. with no free variables. For open terms, the progress theorem fails. This is however not a problem since complete programs—which are the expressions we actually care about evaluating—are always closed.

Independently of the type system and store typing, we should define which state we regard as well-formed. Intuitively, a state is well-formed if the content of the store is consistent with the expression executed by the thread sequence. (We omit global and local counters that are also part of the state, as they are not represented in expressions explicitly.) In case of store π, if there is some evaluation context $\mathcal{E}[\, \texttt{insync}\ l\, e \,]$ in the thread sequence for any verlock location l, then $\pi(l)$ should contain 1, marking that the verlock has been acquired. As for the store σ, containing the content of each reference cell, we may only require that it is well typed.

Definition 7*Suppose π, σ is a well-typed store, and $\overline{f}$ is a well-typed sequence of expressions, where each expression is evaluated by a thread. Then, a state $\pi, \sigma \mid \overline{f}$ is* well-formed*, denoted $\vdash_{wf} \pi, \sigma \mid \overline{f}$, if for each expression f_i ($i < |\overline{f}|$) such that $f_i = \mathcal{E}[\, \texttt{insync}\ l\, e \,]$ for some l, there is $\pi(l) = 1$.*

Of course, a well-typed, closed expression with empty store is well-formed.

According to Lemma 32, the property $\vdash_{wf} \pi, \sigma \mid \overline{f}$ is maintained during evaluation.

Lemma 32 (Well-Formedness Preservation) *If* $\vdash_{wf} \pi, \sigma \mid \overline{f}$ *and* $\pi, \sigma \mid \overline{f} \longrightarrow (\pi, \sigma)' \mid \overline{f}'$ *then* $\vdash_{wf} (\pi, \sigma)' \mid \overline{f}'$.

Proof. Consider a well-formed state $\pi, \sigma \mid e_0$, for some well-typed program $\vdash e_0 : t$ and well-typed store π, σ. Suppose that $e_0 = \mathcal{E}[\,\texttt{sync}\ l\ e\,]$ for some context $\mathcal{E}$, and $\pi(l) = 0$. (Note that when a verlock location l is created, then initially $\pi(l) = 0$ by (R-Lock).) From the latter and premise that the state is well-formed, we know that there is no context $\mathcal{E}'$ such that $e_0 = \mathcal{E}'[\,\texttt{insync}\ l\, e'\,]$ for any e'. From the latter and premise, by (R-Sync), we could reduce expression e_0 to $(\pi, \sigma)' \mid e_1$, such that $e_1 = \mathcal{E}[\,\texttt{insync}\ l\, e\,]$. But then, after reduction step, we have $\pi(l) = 1$ (again by (R-Sync)). Moreover, by type preservation Theorem 7, the new state is well typed. Thus, from the definition of well-formedness, we get immediately that $\vdash_{wf} (\pi, \sigma)' \mid e_1$. Finally, we obtain the needed result by induction on thread creation. □

A state $\pi, \sigma \mid T$ is *deadlocked* if there exist only evaluation contexts $\mathcal{E}$, such that $T = \mathcal{E}[\,\texttt{sync}\ l\ e\,]$ for some verlocks l, such that $\pi(l) = 1$ for each l (i.e. the verlocks are not free) and there is no other evaluation context possible.

Now, we can state the progress theorem.

Theorem 8 (Progress) *Suppose T is a closed, well-typed term (that is, $\Sigma \mid \emptyset; \emptyset; \emptyset \vdash T : t$ for some t and Σ). Then either T is a value or else, for any store π, σ such that $\Sigma \mid \emptyset; \emptyset; \emptyset \vdash \pi, \sigma$ and $\vdash_{wf} \pi, \sigma \mid T$, there is some term T' and store $(\pi, \sigma)'$ with $\pi, \sigma \mid T \longrightarrow (\pi, \sigma)' \mid T'$, or else T is deadlocked on some verlock(s).*

Proof. Straightforward induction on typing derivations. We need only show that either $\pi, \sigma \mid T \longrightarrow (\pi, \sigma)' \mid T'$, or T is a value, or $\pi, \sigma \mid T$ is a deadlocked state. From the definition of $\longrightarrow$, we have $T \longrightarrow T'$ iif $T = \mathcal{E}[\,e_1\,]$, $T' = \mathcal{E}[\,e_1'\,]$, and $e_1 \longrightarrow e_1'$.

Case The variable case cannot occur (because e is closed).
Case The abstract case is immediate, since abstractions are values.
Case $T = e_1\ e_2$ with $\vdash e_1 : t \to^{b,p} s$ and $\vdash e_2 : t$
By the induction hypothesis, either e_1 is a value or else it can make a step of evaluation, and likewise e_2, or T is a deadlocked state. If e_1 can take a step, then $e_1 = \mathcal{E}_1[\,e'\,]$ and $e' \longrightarrow e''$. But then $T = \mathcal{E}[\,e'\,]$ where $\mathcal{E} = \mathcal{E}_1\ e_2$; thus $T \longrightarrow \mathcal{E}[\,e''\,]$. Otherwise, e_1 is a value. If e_1 is a value and e_2 can take a step, then $e_2 = \mathcal{E}_2[\,e'\,]$ and $e' \longrightarrow e''$ then $T = \mathcal{E}[\,e'\,]$ where $\mathcal{E} = e_1\ \mathcal{E}_2$; thus $T \longrightarrow \mathcal{E}[\,e''\,]$. Otherwise, e_1 and e_2 are values, or T is a deadlocked state. Finally, if both e_1 and e_2 are values, then the canonical forms lemma tells us that e_1 has the form $\lambda^{b,p} x : t.\, e_1'$, and so rule (R-App) applies to T.

Other cases are straightforward induction on typing derivations, following the pattern of the case with $T = e_1\ e_2$. □