# Isolation-only Transactions by Typing and Versioning*

Paweł T. Wojciechowski
Ecole Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
Pawel.Wojciechowski@epfl.ch

December 8, 2004

Technical Report IC-2004-104

**Abstract**

In this paper we design a language and runtime support for isolation-only, multithreaded transactions (called tasks) that are intended for time-critical systems. The key concept of our design is the use of a type system to support rollback-free and safe runtime execution of tasks.

We present a first-order type system which can verify information for the concurrency controller. We use an operational semantics to formalize and prove the type soundness result and an isolation property of tasks. The semantics uses a specialized concurrency control algorithm, that is based on access versioning.

We give proofs of type soundness and dynamic correctness of the concurrency control algorithm.

**Key-words:** concurrency, transactions, isolation, type systems

*Research supported by the Swiss National Science Foundation under grant number 21-67715.02 and Hasler Stiftung under grant number DICS-1825.

# Contents



# List of Figures

# 1 Introduction

Multithreading has become an essential part of modern software systems. Although threads simplify the program's conceptual design and allow parallelism on multiple processors, they also increase programming complexity. Programmers must ensure that threads accessing shared data interact correctly, which is notoriously a difficult task. It is natural to ask whether *transactions* [35, 2] could be used; they maintain the illusion of exclusive access to the whole data set while permitting concurrent access at a fine level.

While there have been a variety of implementations of transactions (see [25, 8, 36, 14, 23, 33] among others), comparatively little work has been done on rigorous, language-based approaches to transactions. There are many open questions and challenges: Which standard ACID (Atomicity, Consistency, Isolation, and Durability) properties of transactions are actually useful for common concurrent programming? How should the enforcement of these properties be efficiently implemented? What new language features are required, e.g. for accessing data structures? How much information can be verified statically in order to decrease the runtime support necessary for running transactions?

We consider the above issues in the context of "time-critical" applications which are inherently concurrent; they also demand a high level of robustness and efficiency, with possible timeliness constraints. In the past, such systems were confined mostly to domains like telecommunications switches, flight reservations and air traffic control. However, today more and more systems have similar requirements, including consumer electronics, and mobile embedded systems. Transactions may greatly simplify the development of such systems. Unfortunately, traditional transaction techniques can seldom be transferred from the database to time-critical domain without change; the performance considerations are too different [31, 13, 3].

## 1.1 Design Choices

In this paper we introduce a small language with novel transaction-related constructs, which are suitable for "time-critical" applications. Below we motivate our design choices and the use of a type system, based on surveys [31, 13, 3].

**1. Recoverable executions are not necessary** For critical applications, failures are not tolerable. For those kinds of systems, fault-tolerant techniques such as using redundant hardware and software modules have been developed to reduce the possibility of failure; they are however beyond scope of this paper. We shall use the term *task* for transactions that ensure isolation (and so consistency) only.

Our language has a construct `isolated` $e$ that spawns an isolated task $e$; an implementation in Java is available [29]. The *isolation* property (also known as *serializability*) [35] ensures that any (concurrent) execution of tasks is equivalent to an execution in which the tasks are serialized. The `isolated` construct has been used in the implementation of an example "time-critical" application:

*group communication middleware* [22] for building crash-resilient distributed applications by server replication. It made encoding of the middleware protocols easier and less error-prone [37]. In this paper, we give a rigorous design of this construct, extended with typing for safety guarantees.

**2. Tasks do not include rollback statements** Contrary to databases, task operations may not be recoverable. For instance, tasks of our middleware exchange messages over the network at real time; the messages are then delivered to any distributed applications built on top. Full-scale recovery of tasks is therefore too expensive (it requires distributed agreement [35]) and impractical (since it would also require the applications to be able to rollback their state).

In our previous work [37] we have introduced several novel *versioning* concurrency control algorithms for scheduling access by tasks to shared data with the isolation guarantee. The algorithms are rollback-free. They have been used to implement the `isolated` construct. A distinct feature of the algorithms is that tasks can be themselves multithreaded; we motivate this feature below.

The versioning algorithms require however some language support since all resources accessed by tasks must be known *a priori*. To make the language safe, we therefore propose a type system that can verify data declared for the concurrency controller; to our knowledge it is the first presentation of a type system that supports multithreaded tasks. The type system builds on the type system for detection of race conditions proposed by Flanagan and Abadi [9].

**3. Tasks must be efficient** Roughly, a greater degree of concurrency leads to higher performance. The degree depends on the amount of information available to the concurrency controller. Our concurrency control algorithms allow fine-grain scheduling of *arbitrary* task operations, such as data read/write and network message outputs. These operations just need to be protected by *versioning locks* (or *verlocks*) for isolation-preserving scheduling; we will show the use of verlocks in §2. Alternatively, (ver)locking could be left as an implementation issue of the `isolated` construct. In this paper, however, we keep verlocks as a language construct, as it allows us to explain the semantics of the versioning algorithms and typing at the level of detail that is required for rigorous proofs of isolation. It has also simplified our experimental implementation of `isolated`.

Knowing more information about intended program behaviour, such as predefined patterns of acquiring verlocks, allows one to select a versioning algorithm that uses more efficient synchronization techniques [37]. We propose in this paper that the pattern declaration should be verified statically, using a type system. We explain this idea using a *Basic Versioning Algorithm* (BVA) – one of the simplest versioning algorithms possible. It permits less concurrency than its two optimized variants described in [37], but it requires a simple type system that only verifies declarations of verlock names. This is sufficient to illustrate our novel *hybrid approach* that combines concurrency control with typing.

We use Flanagan and Abadi's approach in [9] to ensure that *all* shared data accesses are protected by verlocks. This guarantee could be relaxed in the future, e.g. objects known to be immutable need not be visible to the concurrency controller when accessed, and so they could be left unprotected.

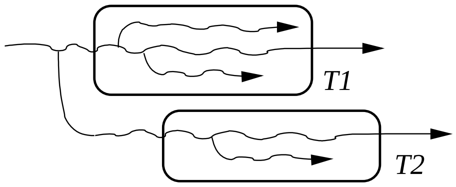


Figure 1: Concurrent, multithreaded transactions

**4. Serializability is sometimes too conservative** Several authors describe the limitations of serializability as a correctness criterion [21, 31]. To support cases when isolation is too conservative, we allow tasks to be multithreaded. Threads are lightweight processes that communicate using shared mutable data and synchronize (in the scope of a task) by acquiring and releasing verlocks; other synchronization means such as monitors could be also used. Individual threads may fork and e.g. start other tasks. Figure 1 illustrates two concurrent, multithreaded tasks $T1$ and $T2$. Execution of each task is atomic with respect to other tasks. We do not require however threads within a task to be serializable; thus, they can engage in two-way communication using shared data.

We have found multithreaded tasks useful in the implementation of concurrent protocols, in which a network message must be processed by a set of cooperating threads [37]. Protocols may spawn threads, e.g. for better use of multi-CPU architectures, or for slow I/O operations such as disk access. Multithreaded tasks guarantee however that each message is processed by these concurrent threads using a consistent set of protocol session or message-specific data. (Note that constructs such as *nested transactions* [35] do not apply here, as they normally do not relax isolation between subtransactions, and they depend on rollback-recovery.)

The decision to allow multithreaded tasks means however, that in our language it may not be possible to verify the isolation property statically (at compile time only), since the language allows threads to be created and terminated dynamically at will. This, together with the requirements of rollback-freedom and language safety, motivates our hybrid, type-directed approach to concurrency control.

The previous work closest to our own is Flanagan and Qadeer's [10] type system for specifying and verifying the atomicity of methods in multithreaded Java programs (the notion of "atomicity" is equivalent to serializability in this paper). Harris and Fraser [15] have been also investigating an extension of Java with atomic code blocks; we describe this and other related work in §5. However, atomic code fragments must be sequential, while our tasks can be multithreaded. Moreover, our applications may demand different levels of performance and isolation constraints; these varying demands will lead to a multiplicity of runtime concurrency controllers, based on a variety of scheduling algorithms. It could be however interesting to investigate the possibility of adding multiple threads within atomic blocks of [10, 15] using the approach described in this paper.

### 1.2 Contributions

We make several contributions:

- We have shown several results and theorems about our *type-directed approach* to concurrency control of multithreaded tasks; the main result is that well-typed programs satisfy the isolation property;
- We present an *operational semantics* of tasks; the semantics has been split into a dynamic semantics of the host language constructs, and of the concurrency controller. The former explains program evaluation, while the latter defines isolation with a specific scheduling algorithm;
- We have used the semantics to formalize and prove correct the BVA algorithm; to our best knowledge it is the first rigorous proof of isolation preservation in multithreaded tasks, which makes data accesses explicit.

The paper is organized as follows. §2 explains the constructs using an example program. §3 – the heart of our paper – defines syntax, semantics, and typing of the calculus, including a simple concurrency control algorithm. §4 states the main results, including type soundness and dynamic correctness of the concurrency control algorithm. §5 discusses related work and §6 concludes. Proofs are in the Appendix.

## 2 Example

Consider a central air route surveillance station, which controls air traffic in a large geographic area (the example is adopted from [31]). It receives data from each local station and records them in a corresponding "track table" (the record of a single aircraft is called a "track"). Below are two code fragments (assumed to be concurrent) expressed using our language.

```
newlock x:RegionA in
newlock y:RegionB in

isolated x,y   (* task T1: hand-over *)
{
  sync x regionA.withdraw(aircraft);
  sync y regionB.deposit(aircraft);
}

isolated x,y   (* task T2: control *)
{
  aircraft_tabA := sync x regionA.get();
  aircraft_tabB := sync y regionB.get();
  analyseAndReport(aircraft_tabA, aircraft_tabB);
}
```

Task $T1$ is recording aircraft movement based on information from two adjacent local stations; the new correlation is stored in track tables of the corresponding regions. It must maintain a consistent view, i.e. a track must not disappear or appear in more than one table.

Suppose some aircraft moved from region $A$ to $B$ while task $T1$ is updating the track tables. Meanwhile, a task $T2$ analyses the traffic pattern in the controlled area and produces a warning if two aircraft fail to maintain minimum separation. $T2$ must obtain a snapshot view of the controlled area by reading the tables. $T2$ could obtain an inconsistent view if it first retrieves data in region $B$ before $T1$ updates it. This may lead to failure to prevent an impending collision if the aircraft moved from $A$ to $B$ is missing.

The isolation property of tasks ensures however that any (concurrent) execution of $T1$ and $T2$ is equivalent to an execution in which the tasks are serialized. This means that they will never interfere.

Execution of `newlock` $x : t$ `in` $e$ creates a new verlock $x$ (or lock in short) of type $t$; the lock type identifies data protected by the lock. The expression `sync` $e\ e'$ is similar to Java's `synchronized` statement [12], i.e. the expression $e$ is evaluated first, and should yield a lock, which is then acquired when possible; the expression $e'$ is then evaluated; and finally the lock is released.

Execution of `isolated` $\overline{e}\ e$ creates a new task for the evaluation of expression $e$. After the creation, $e$ commences execution, in parallel with the rest of the body of the spawning program. Tasks can spawn their own threads, e.g. $T2$ could spawn a new thread that updates $T2$'s table with some emergency report. The declaration $\overline{e}$ should give verlocks that can be used by the task to control access to shared data. We assume that information on locks is provided explicitly, and leave type inference as an open problem.

Flanagan and Abadi's type system provides guarantees for the concurrency scheduler that all such data accesses are made using verlocks. Our extension of their type system also verifies if verlocks that may be acquired by a task are known before the task commences, i.e. they are declared in $\overline{e}$. It thus eliminates errors due to omission of such declarations, e.g. the above program does not typecheck if the arguments $x$ or $y$ of `isolated` are removed. The above two guarantees enable a safe use of abort-free versioning algorithms.

Execution of tasks $T1$ and $T2$ satisfies the isolation property. However, any threads *inside* tasks are not constrained; a required synchronization policy could be encoded using verlocks (accompanied in the scope of a task with any other synchronization means if needed).

# 3 Language for Isolated Tasks

## 3.1 Syntax

We define our language as the call-by-value $\lambda$-calculus, extended with reference cells, isolated tasks, and versioning locks. The abstract syntax is in Figure 2. 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.).

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

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

Figure 2: The *iso*-calculus: Syntax

**Types** Types include the base type `Unit` of unit expressions, which abstracts away from concrete ground types for basic constants (integers, Booleans, etc.), the type $t \rightarrow^{a,p} t$ of functions, the type $\texttt{Ref}_m\, t$ of reference cells containing a value of type $t$, and finally a singleton lock type $m$. A *singleton lock type* is the type of a single lock. The types of references and functions are decorated by correspondingly, $m$ and $a, p$, where $m$ is a singleton lock 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 lock 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.

**Values and Basic Expressions** 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 lock 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).

**Threads and Tasks** The 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. Execution of `isolated` $\overline{e}$ $e$ creates a new isolated task thread for the evaluation of expression $e$. Tasks can use `fork` to spawn their own threads. The declaration $\overline{e}$ should give verlocks that can be used by a task to control access to shared data. All program threads will be interleaved while providing the illusion that tasks are executed in isolation.

**Verlocks** The execution of `newlock` $x : m$ `in` $e$ creates a new unique name $x$ of a versioning lock (or verlock). It also introduces the type variable $m$ which denotes the singleton lock type of the newly created verlock. Both $x$ and $m$ may be referred to in the expression $e$, i.e. $x$ and $m$ are bound in $e$. The expression `sync` $e\ e'$ is similar to Java's `synchronized` statement [12], i.e. the expression $e$ is evaluated first, and should yield a verlock, which is then acquired when possible; the expression $e'$ is then evaluated; and finally the verlock is released. Verlocks combine a simple lock (mutex) for protection against simultaneous data accesses by concurrent threads, with an *access versioning* algorithm that schedules lock acquisitions by (threads of) isolated tasks based on access versions; the details of the algorithm will be given in §3.2.

## 3.2 Operational Semantics

We specify the operational semantics using the rules defined in Figures 3 and 4. A state $S$ consists of three elements: a lock store $\pi$ and a reference store $\sigma$, which are sometimes referred to collectively as a store $\pi, \sigma$, and a collection of expressions $T$, which are organized as a sequence $T_0, ..., T_n$. Each expression $T_i$ in the sequence represents a *thread*.

The *lock store* $\pi$ is a finite map from lock locations to their states; a lock location has two states, unlocked (0) and locked (1), and is initially unlocked. The *reference store* $\sigma$ 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 §3.1, extended with a new construct `task` $pv\ T$. The construct is not part of the language to be used by programmers; it will used later to explain semantics.

We define a small-step evaluation relation $\pi, \sigma \mid e \longrightarrow \pi', \sigma' \mid e'$, read "expression $e$ reduces to expression $e'$ in one step, with stores $\pi, \sigma$ being transformed to $\pi', \sigma'$". We also use $\longrightarrow^*$ for a sequence of small-step reductions. By *concurrent evaluation*, or *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 place according to the transition rules in Figure 3 and 4. 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 $eval(e, v_0)$ is therefore a relation, not a partial function.

Below we describe reduction rules in Figure 3. These rules are common for all versioning concurrency control algorithms, while rules in Figure 4 (described later) define our example algorithm.

The first four evaluation rules are the standard rules of a call-by-value $\lambda$-calculus [27], extended with references. We write $\{v/x\}e$ to denote the capture-free substitution of $v$ for $x$ in the expression $e$. The notation $(\sigma, r \mapsto v)$ means "the store that maps $r$ to $v$ and maps all other locations to the same thing as $\sigma$". 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 $r$. For instance, let us look at the rule (R-Assign). We use the notation $\sigma[r \mapsto v]$ to denote update of map $\sigma$ 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 describe the *Basic Versioning Algorithm* (BVA) for "isolated evaluation" of tasks. The algorithm consists of four evaluation rules (BVA-0–3) in Figure 4, which define creation and destruction of tasks, and verlock acquisition and release. It is one of the simplest algorithms possible. The semantics can be however easily extended to more concurrent versioning algorithms of [37].

**BVA: Task Creation and Destruction** The program state is extended with a map $gv$ of global version counters $gv(l)$ for each lock $l$ in $\pi$ (initialized to 0). A *version* is an integer playing a rôle of access capability. Each lock $l$ maintains a local version counter $lv(l)$, which is also initialized to 0; a map $lv$ of local counters is part of the state, too. For clarity we usually omit the counters in the rules when possible. The algorithm maintains an invariant (R-Invar) that a local version of each lock is equal or less than a global version of the lock, and it is equal or greater than zero.

Evaluation of a term `isolated` $\bar{l}$ $e$ creates a new thread for evaluation of expression `task` $pv$ $e$; see (R-Isol). The term `task` $pv$ $e$ is a *task* evaluating expression $e$, where $pv$ is a *private versions* map of (ver)locks $\bar{l}$ *declared* by term `isolated`. The map $pv$ associates lock locations with globally unique versions, maintained by global version counters $gv$. The map $pv$ is created for a given set of (ver)locks dynamically in one atomic step, and remains constant for the task's

**State Space**

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

**Evaluation Contexts**

$$
\mathcal{E} \;=\; [\,] \mid \mathcal{E}\ e \mid v\ \mathcal{E} \mid \mathtt{ref}_m\, \mathcal{E} \mid\, !\mathcal{E} \mid \mathcal{E} := e \mid r := \mathcal{E} \mid \mathtt{sync}\ \mathcal{E}\ e \mid \mathtt{insync}\ l\, \mathcal{E}
$$
$$
\mid \mathtt{isolated}\ \overline{l}\mathcal{E}\overline{e}\ e \mid \mathtt{task}\ pv\ \mathcal{E} \mid \mathcal{E}, T \mid T, \mathcal{E}
$$

**Structural Congruence**

$$
T, T' \equiv T', T \qquad T, () \equiv T \qquad \frac{\pi, \sigma \mid f \longrightarrow \pi', \sigma' \mid f'}{\pi, \sigma \mid \mathcal{E}[\, f\,] \longrightarrow \pi', \sigma' \mid \mathcal{E}[\, f'\,]} \qquad \frac{f \longrightarrow f'}{\pi, \sigma \mid f \longrightarrow \pi, \sigma \mid f'}
$$

**Transition Rules (Part I)**

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

$$
\lambda x.\, e\ v \longrightarrow e\{v/x\} \tag{R-App}
$$

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

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

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

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

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

$$
\frac{\pi(l) = 1}{\pi, \sigma \mid \mathtt{insync}\ l\, v \longrightarrow \pi[l \mapsto 0], \sigma \mid v} \tag{R-InSync}
$$

Figure 3: The *iso*-calculus: Reduction semantics

**Transition Rules (Part II)**

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

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

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

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

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

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

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

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

Figure 4: The *iso*-calculus: Rules of Basic Versioning Algorithm (BVA)

lifetime. Program evaluation maintains an invariant that a private version of each lock in a private versions map of every task is globally unique.

Tasks are analogous to multithreaded transactions decomposed to ensure an isolation property only. Tasks can spawn their own threads using `fork`; see (R-Fork'). Tasks are used only for their side-effects, which are in our case modifications to the store. Once expression $e$ of `task` $pv\ e$ yields a value and $pv(l) - 1$ equals $lv(l)$ for all $l$ in $dom(pv)$, the task returns an empty value (); see (R-Task). Then we say that the task has *completed* or *terminated*.

**Serialized and Isolated Evaluation** Two tasks are executed *serially* if one task commences after another one has completed. By *serialized evaluation*, or *serial run*, we mean evaluation, in which all tasks are executed serially. (Note that a serial run is also concurrent since serialized tasks may be themselves multithreaded.)

A state $S$ is *task-free* if it does not have a context $\mathcal{E}[\,\texttt{task}\ pv\ T\,]$. Any task-free state is called a *result state*. Note that the result states subsume data stored in all reference cells. An *effect* in our language is any change to the content of reference cells.

We define *isolation* to mean, intuitively, that the effects of one task are not visible to other tasks executing concurrently; from the perspective of a task, it appears that tasks execute sequentially rather than in parallel. Isolation has been proposed as the correctness condition of concurrency control algorithms [2]. We have also used it to define consistency of message processing in concurrent protocols [37].

The isolated execution of tasks is captured precisely using the notion of noninterference. Tasks in a concurrent run *do not interfere* (or satisfy the *noninterference* property) if there exists some ideal serial run $R^{\mathrm{s}}$ of all the tasks, such that given any reference, the order of accessing the reference by tasks in the concurrent run is the same as in $R^{\mathrm{s}}$.

**Definition 1** (Isolation Property) Evaluation of an expression $e$ satisfies an *isolation property* if all tasks of $e$ do not interfere. A *program* satisfies the isolation property if all terminating evaluations of the program satisfy this property.

**BVA: Verlock Acquisition and Release** The expression (R-Lock) dynamically creates a new verlock's lock location $l$ (with the initial state 0) and replaces 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 lock type. A lock store $\pi$ that binds a verlock's lock 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 lock location, where it is clear from the context what we mean.

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

The semantics of `sync` $e\ e'$ executed by a task is defined by rule (R-Sync). The expression $e$ is evaluated first, and should yield a verlock $l$, which is then acquired if free *and* if the task holds a version number $pv$ for $l$ that matches a local version maintained by $l$ (i.e. $pv(l) - 1 = lv(l)$). The expression $e'$ is then evaluated as part of an expression `insync` $l\,e'$. The verlock is released by (R-InSync) when the expression $e'$ reduces to a value $v$ (then `insync` $l\,v$ is replaced by $v$).

The second premise of rule (R-Sync) ($pv(l)-1 = lv(l)$) guarantees that a task can acquire a verlock only at a time when it is *safe*, i.e. when accessing data protected by the verlock does not invalidate isolation. Otherwise, the verlock's lock is not taken even if it is free, resulting in the task being blocked. However, each lock will be eventually acquired (evaluation progress) if only tasks are themselves deadlock-free and terminate.

Task deadlock can be avoided by imposing a strict partial order on verlocks within each task, and respecting this order when acquiring verlocks; our language and type system can be extended with this principle by embodying the solution described in [9].

**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 task are known at a time when the task is spawned, so that the (R-Isol) rule can create the private version for each such verlock type, stored in the task's map $pv$. To maximize parallelism, we require only such verlocks to be declared. In §4, we show that both conditions are verified statically by the type system in §3.3.

## 3.3 Typing

The type system is formulated as a deductive proof system, defined using conclusions (or judgments) and the static inference rules for reasoning about the judgments in Figure 5. The typing judgment for expressions has the form $\Gamma; a; p \vdash e : t$, read "expression $e$ has type $t$ in environment $\Gamma$ with allocation $a$ and permission $p$", where an environment $\Gamma$ is a finite mapping from free variables to types; type variables are bound to a kind `Lock`. 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: (*i*) the current thread holds at least the verlocks described by $p$ (Condition 1), (*ii*) if $e$ is part of a task, then the task has declared all verlocks described by $a$ (Condition 2), and (*iii*) the free variables of $e$ are given bindings consistent with $\Gamma$. We will show in §4 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 conditions [9]. 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 [9]. The set of typing rules in Figure 5 has

**Judgments**

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

**Typing Rules**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 5: The first-order type system for the *iso*-calculus

been obtained by extending this approach with allocations needed to verify Condition 2, and adding a new rule for typing the `isolated` construct. 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 with permissions in [9], 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 lock 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 lock 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 $e''$. As in [9], the corresponding rules (T-Deref) and (T-Assign) check if a singleton lock type $m$ decorating the reference type is among lock types mentioned in the current permission $p$. The permission $p$ can be extended with $m$ only while typechecking a synchronization expression `sync` $e\ e''$, where $e$ has type $m$ (see typing of $e$ in (T-Sync)).

To verify if a task $e_0$ executing `sync` $e\ e'$ 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 `isolated` that has spawned task $e_0$. The rule (T-Isol) creates the allocation $a$ from singleton types of all verlocks declared by the task; the allocation is then used for typechecking the body of the task.

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 tasks to be multithreaded, while permissions are nulled since spawned threads do not inherit locks 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.

# 4 Type System Results

The fundamental property of the type system and abstract machine of our language is that evaluation of well-typed, terminating programs satisfies the isolation property. 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**

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

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

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

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

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

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

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

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

Figure 6: Additional judgments and rules for typing states

$Exp_{ext}$, and then to states as shown in Figure 6. 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 $\pi, \sigma$. These types have been collected as a *store typing* $\Sigma$ – a finite function mapping locations to types, and type variables to kinds.

Type preservation and progress yield that our type system is *sound*. 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 task spawned using the `isolated` construct, then the task has a private version of this verlock (which is possible only if the name of it has been specified as 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 prove in Appendix that evaluation of well-typed, terminating programs satisfies the isolation property.

Below we state formally the absence of race conditions and the absence of non-declared verlocks properties. Finally, we give our main result of isolation preservation in §4.3.

## 4.1 Flanagan and Abadi's Absence of Races

After removing allocations $a$ and the rule (T-Isol) for typing the construct `isolated` in Figure 5, and replacing the semantics of verlocks by simple locks, we obtain Flanagan and Abadi's first-order type system [9]. The fundamental property of this type system is that well-typed programs do not have race conditions. Below are Lemmas as found in [9], extended with store typing $\Sigma$ and allocations.

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 $\emptyset, \emptyset \mid e \longrightarrow^* S$ and $S$ has a race condition.

Independently of the type system, locks provide mutual exclusion, in that two threads can never be in a critical section on the same lock. An expression $f$ is in a *critical section* on a lock 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 lock in $S$. According to Lemma 1, the property $\vdash_{cs} S$ is maintained during evaluation.

**Lemma 1 (Mutual Exclusion [9])** 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 lock.

**Lemma 2 (Lock-Based Protection [9])** Suppose that $\Sigma \mid \Gamma; a; p \vdash f : t$, and $f$ accesses reference location $r$. Then $\Sigma \mid \Gamma; a; p \vdash r : \texttt{Ref}_m\, t'$ for some lock type $m$ and type $t'$. Furthermore, there exists lock 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 [9])** 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 [9])** If $\vdash e : t$ then $e$ does not have a race condition.

## 4.2 Absence of Non-declared Verlocks

An expression $f$ is *part of* a task `task` $pv\ T$ if $T = \mathcal{E}[\,f\,]$ for some evaluation context $\mathcal{E}$. A task `task` $pv\ T$ *has a version* of a lock $l$ if $pv(l)$ is defined. An expression $f$ *has a version* of a lock $l$ if there exists some task which has a version of $l$, and $f$ is part of this task. An expression $f$ *requests* a lock location $l$ if $f = \mathcal{E}[\,\texttt{sync}\ l\ e\,]$ for some evaluation context $\mathcal{E}$ and expression $e$. A task `task` $pv\ T$ is in a *critical section* on a lock location $l$, if some thread of $T$ is in a critical section on the lock location $l$.

Now, for the complete language with `isolated` and `task`, the judgment $\vdash_{cs} S$ says in addition to mutual exclusion property stated in §4.1, that each task being in a critical section on some lock in state $S$ has a version of this lock (see Figure 7). 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 $\vdash_{cs} S'$.

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 lock location $l$. Then $\Sigma \mid \Gamma; a; p \vdash l : m$ for some lock type $m$. Furthermore, there exists a task `task` $pv\ T$ which $f$ is part of, such that $\Sigma \mid \Gamma; a; p \vdash \texttt{task}\ pv\ T : \texttt{Unit}$ and version $pv(l)$ is defined.

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

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

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

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

**Judgments**

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

**Rules for Critical Sections of [9]**

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

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

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

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

**Additional Rules for Critical Sections and Tasks**

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

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

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

Figure 7: Judgments and rules for reasoning about critical sections and tasks

### 4.3 The Main Result of Isolation Preservation

We have defined the isolated evaluation for complete tasks (see §3.2). 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 task-free state. The judgment for such states has the form $\vdash_{tf} S$, read "state $S$ is well-formed and task-free", which means that either no task 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 $S'$.

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

**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.

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).

Proof of Theorem 3 is based on dynamic correctness of the BVA algorithm, formulated using the following theorem.

**Theorem 4 (Noninterference)** If a program has properties $(i)$ and $(ii)$ (see §4, 2nd paragraph) then any evaluation of the program up to any result state, using the BVA algorithm, satisfies the noninterference property.

### 4.4 Proving 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, 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. Below are excerpts from the proof; the complete proof has been included in Appendix.

**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 $\pi, \sigma$ to be well typed. (For clarity, we omit permissions $p$ from the context.)

**Definition 2** A store $\pi, \sigma$ is said to be *well typed* with respect to a store typing $\Sigma$ and a typing context $\Gamma$, 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 $\pi, \sigma$ is consistent with a store typing $\Sigma$ if every value in the store has the type predicted by the store typing.

Type preservation for our language states that the reductions defined in Figures 3 and 4 preserve type:

**Theorem 5 (Type Preservation)** If $\Sigma \mid \Gamma; a \vdash T : t$ and $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ and $\pi, \sigma \mid T \longrightarrow (\pi, \sigma)' \mid T'$, then for some $\Sigma' \supseteq \Sigma$, $\Sigma' \mid \Gamma; a \vdash T' : t$ and $\Sigma' \mid \Gamma; a \vdash (\pi, \sigma)'$.

**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.

We also had to show 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).

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 $\pi$, if there is some evaluation context $\mathcal{E}[\,\texttt{insync}\; l\, e\,]$ in the thread sequence for any lock location $l$, then $\pi(l)$ should contain 1, marking that the lock has been acquired. As for the store $\sigma$, containing the content of each reference cell, we may only require that it is well typed.

**Definition 3** Suppose $\pi, \sigma$ 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 7, the property $\vdash_{wf} \pi, \sigma \mid \overline{f}$ is maintained during evaluation.

**Lemma 7 (Well-Formedness Preservation)** If $\vdash_{wf} \pi, \sigma \mid \overline{f}$ and $\pi, \sigma \mid \overline{f} \longrightarrow (\pi, \sigma)' \mid \overline{f}'$ then $\vdash_{wf} (\pi, \sigma)' \mid \overline{f}'$.

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 6 (Progress)** Suppose $T$ is a closed, well-typed term (that is, $\Sigma \mid \emptyset; \emptyset; \emptyset \vdash T : t$ for some $t$ and $\Sigma$). Then either $T$ is a value or else, for any store $\pi, \sigma$ 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 lock(s).

# 5 Related Work

There have been recently many proposals of concurrent languages with novel synchronization primitives, e.g. Polyphonic C# [1], which is based on the join-pattern abstraction [11], and Concurrent Haskell [20], Concurrent ML [24], Pict [26] and Nomadic Pict [30] which have synchronization constructs based on channel abstractions. They enable to express complex synchronization code more easily than when using standard constructs, such as monitors and locks. This work is orthogonal to the goals of this paper. We are primarily focused on high-level transactional facilities that provide automatic concurrency control.

The work in this paper builds on research in two areas: language and system support for isolation, and the semantics of concurrency control. Below we discuss example work in these two areas, focusing on atomic transactions, lock-free atomic blocks, concurrency control, and transaction models.

**Atomic Transactions** Atomic transactions that can be decomposed to satisfy only a subset of the Atomicity, Consistency, Isolation, and Durability (ACID) properties appeared in distributed operating systems, such as Camelot [8], in modern transactional platforms, such as Sun Enterprise JavaBeans (EJB) [33] and Microsoft Transaction Server (MTS) [23], and in programming languages, such as Arjuna [25], Avalon/C++ [6] and Venari/ML [14, 36]. Venari/ML is an extension of the ML programming language with atomic transactions. Concurrency control is factored out into a separate mechanism that the programmer can use to ensure isolation. Higher-order functions in ML allow the programmer to easily express transactions with desirable ACID properties. Transactions can be multithreaded. However, the design choices were not defined formally, and the details remain a matter of implementation.

**Lock-Free Atomic Blocks** While our construct `isolated` can allow to declare multithreaded sections of code to be executed in isolation, several researchers have proposed programming language features for isolation of sequential code blocks. For instance, Flanagan and Qadeer [10] proposed a type system for specifying and verifying the atomicity of methods in multithreaded Java programs, where the notion of "atomicity" is similar to linearizability [16] for concurrent objects, and isolation in this paper. Their approach allows program methods to be annotated with the keyword `atomic`. If the program type checks, then any interaction between an atomic method executed by a thread and steps of other threads is guaranteed to be benign, in the sense that these interactions do not change the program's overall behaviour. The type system is a synthesis of Lipton's theory of left and right movers (for proving properties of parallel programs) and type systems for race detection.

More recently, Harris and Fraser [15] have been investigating an extension of Java with (again, sequential only) atomic code blocks that implement Hoare's conditional critical regions (CCRs) [17]. The programmer can guard a conditional region by an arbitrary boolean condition, with calling threads blocking until the guard is satisfied. The implementation is based on mapping CCRs onto a *software transactional memory* which groups together series of memory accesses and makes them appear atomic.

Contrary to the above work on atomicity, our isolated tasks can be multithreaded, and the task implementation depends on scheduling provided by concurrency control algorithms. It will be however interesting to examine if the non-blocking concurrent data structures could be integrated with our dynamic scheme for isolation of groups of threads.

There have been also some approaches to solve synchronization problems at the hardware level. For instance, Rajwar and Goodman [28] have proposed a technique that uses hardware to convert lock-based critical sections into lock-free optimistic transactions.

**Concurrency Control** The concurrency control algorithms that are used to implement atomic transactions in database systems 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 the literature can be found in [2]. 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. [32].

The BVA algorithm has some resemblance with *two-phase locking* [2, 35]. 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. Execution of the BVA algorithm orders conflicting operations of tasks according to versions, like in the *timestamp* algorithms [2, 35]. However, we associate versions with verlocks, not with transactions. Therefore all data accesses protected by verlocks can be made in the right order for the isolation property (the verlock requests with too high versions are simply delayed), unlike basic timestamp algorithms for 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)[2].

Methods of *deadlock avoidance* in allocating resources [32] are also relevant to our work. For instance, the *banker's algorithm* (introduced by Dijkstra [7]) 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* [18] 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 (i.e. verlocks) before it can commence. However, the history of verlock acquisitions by different tasks is always acyclic since versions impose a total order on the acquisitions. Since tasks are assumed to complete, old versions will be eventually upgraded. Moreover, accesses to data protected by verlocks are performed 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.

**Transaction Models** Turning to the semantics of transactions, Chrysanthis and Ramamritham [5] have specified the broad spectrum of transactional models, including nested transactions. More recently, Black *et al.* [4] have defined an equation theory of operators, where an operator corresponds to an individual ACID property. The operators can be composed, giving different semantics to transactions. These models are however presented abstractly, without being integrated with any language or calculus.

Vitek *et al.* [34] and Jagannathan and Vitek [19] have recently proposed a calculi-based model of standard ACID transactions. They have formalized the optimistic and two-phase locking concurrency control strategies. Their approach to formalization of the isolation property (I) is similar to the one in this paper. However, the soundness result rests upon an abstract notion of permutable actions, while our soundness result and proofs make explicit data accesses and task noninterference.

# 6 Conclusion and Future Work

Our main result in this paper is that well-typed programs with multithreaded tasks satisfy the *isolation property*. For clarity, we have therefore chosen a somewhat idealised concurrency control algorithm. The algorithm is not free from drawbacks. For instance, if a thread is preempted while holding a lock then no other thread can safely access the lock. In the future, we would like to extend the approach to the algorithms of [37] that admit more concurrency. We could add distinction between read-only and read-write locking for efficiency. It may be also worthwhile to investigate algorithms for inferring the typing annotations.

**Acknowledgments** We thank Olivier Rütti for his implementation work on the SAMOA protocol framework and isolated tasks. We also thank other members of the Crystall project for discussions. Olivier and Peter Sewell provided comments on this paper.

# A Well-typed Programs Satisfy Isolation

## A.1 Absence of Non-declared Verlocks

The type system provides rules for proving that in well-typed programs: $(i)$ each task spawned using the `isolated` construct can only read or update a reference which is protected by a verlock, and $(ii)$ the verlock itself was specified in the argument of `isolated`. Since by the absence of race conditions (Theorem 1 in §4.1, derived from [9]), in well-typed programs a task cannot access a reference without first obtaining a verlock, we only need to show the second part of the above result. Below we use the semantics to state this property formally.

An expression $f$ is *part of* a task $\texttt{task}\ pv\ T$ if $T = \mathcal{E}[\,f\,]$ for some evaluation context $\mathcal{E}$. A task $\texttt{task}\ pv\ T$ *has a version* of a lock $l$ if $pv(l)$ is defined. An expression $f$ *has a version* of a lock $l$ if there exists some task which has a version of $l$, and $f$ is part of this task. An expression $f$ *requests* a lock location $l$ if $f = \mathcal{E}[\,\texttt{sync}\ l\ e\,]$ for some evaluation context $\mathcal{E}$ and expression $e$. A task $\texttt{task}\ pv\ T$ is in a *critical section* on a lock location $l$, if some thread of $T$ is in a critical section on the lock location $l$.

Now, for the complete language with `isolated` and `task`, the judgment $\vdash_{cs} S$ says in addition to mutual exclusion property stated in §4.1, that each task being in a critical section on some lock in state $S$ has a version of this lock (see Figure 7). 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 $S = \pi, \sigma \mid \mathcal{E}[\,\texttt{task}\ pv\ T\,]$ for some well-typed store $\pi, \sigma$, context $\mathcal{E}$ and (possibly multithreaded) term $T$. By rule (R-Task) and evaluation context for `task`, we know that task $\texttt{task}\ pv\ T$ can either reduce to the empty value $()$ if $T$ is a value, or to $\texttt{task}\ pv\ T'$ otherwise, where $T'$ is some expression. The former case is trivial since we have immediately

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

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

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

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

for some $\mathcal{M}$ by (CS-State) and the fact that task $\texttt{task}\ pv\ T$ is a thread in $S$ (by (R-Isol)). 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 lock $l'$. 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 `task` $pv\ T'$ does not have a version of lock $l'$, i.e. $pv(l')$ is not defined. But this is not possible, since by version-based protection Lemma 5 (below), if a task `task` $pv\ T$ requests lock 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, $\mathcal{M}' \vdash_{cs} \texttt{task}\ pv\ T'$ 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 $\mathcal{M} \vdash_{cs} \texttt{task}\ pv\ T'$ 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 1 (in §A.3.1) and (CS-State) and induction on threads in $S$. $\square$

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 lock location $l$. Then $\Sigma \mid \Gamma; a; p \vdash l : m$ for some lock type $m$. Furthermore, there exists a task `task` $pv\ T$ which $f$ is part of, such that $\Sigma \mid \Gamma; a; p \vdash \texttt{task}\ pv\ T : \texttt{Unit}$ and version $pv(l)$ is defined.

Proof. If $f$ requests a lock location $l$ then from the definition of "requesting a lock 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 lock 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 task with allocation $a$ (since $a \neq \emptyset$).

Hence, by (T-Isol) $f$ is reduced from some expression `isolated` $\bar{l}\ e_0$, such that $\Sigma \mid \Gamma; a'; p' \vdash \texttt{isolated}\ \bar{l}\ e_0 : \texttt{Unit}$ (for some $a'$ and $p'$), where $\bar{l}$ is a sequence of lock locations. Moreover, since allocation $a$ is preserved during task 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-Isol) expression $e_0$ can only reduce to `task` $pv\ e_0$ 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-Task), term `task` $pv$ $T$ has type `Unit`, which completes the proof. □

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

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

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

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

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

## A.2 The Main Result of Isolation Preservation

We have defined the isolated evaluation for complete tasks (see §3.2). 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 that begin and finish in a task-free state. The judgment for such states has the form $\vdash_{tf} S$, read "state $S$ is task-free", which means that either no task 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 task in $S$ (if we would let $S$ not to be task-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 tasks 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 B.1).

Moreover, from $\vdash_{cs} S$, the lock-based protection Lemma 2 and mutual exclusion Lemma 1 give another requirement (i.e. Property 1, see B.1) 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 tasks. Hence, by noninterference Theorem 6 (that we prove in §B) and the definition of isolation, we obtain the needed result. $\Box$

We conclude that well-typed, terminating programs satisfy the isolation property. A program is *terminating* if all its runs terminate; a run *terminates* if it reduces to a value.

**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 $\pi, \sigma$, that is $\Sigma \mid \emptyset; \emptyset; \emptyset \vdash \pi, \sigma$ for some $\Sigma$. Then $\vdash \pi, \sigma \mid e : t$ by Definition 2 (see §A.3.1) 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 task by syntax (see Figure 2). Pick up any terminating trace such that $\pi, \sigma \mid e \longrightarrow^* \pi', \sigma' \mid v_0$ for some store $\pi', \sigma'$ 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 $v_0 \neq \texttt{task}\ pv\ T$ 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. $\Box$

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).

## A.3 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, 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.3.1 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 $\pi, \sigma$ 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 2** A store $\pi, \sigma$ is said to be *well typed* with respect to a store typing $\Sigma$ and a typing context $\Gamma$, 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 $\pi, \sigma$ is consistent with a store typing $\Sigma$ if every value in the store has the type predicted by the store typing.

By canonical forms (Lemma 14 in §A.3.2), each *location value* $l \in dom(\pi, \sigma)$ can be either a lock location, or a reference location, depending on a concrete type. For simplicity, we often refer to $\pi, \sigma$ as the store, meaning individual stores, i.e. either $\pi$ or $\sigma$, depending on a given value and type. If a location value $l$ is a lock location then it is kept in a lock store $\pi$; if the value is a reference location then it is kept in a reference store $\sigma$.

Type preservation for our language states that the reductions defined in Figures 3 and 4 preserve type:

**Theorem 4 (Type Preservation)** If $\Sigma \mid \Gamma; a \vdash T : t$ and $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ and $\pi, \sigma \mid T \longrightarrow (\pi, \sigma)' \mid T'$, 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 $\Sigma$ on the values of all the old locations) such that a new term $T'$ is well typed with respect to $\Sigma'$. This new store typing $\Sigma'$ is either $\Sigma$ 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** $\pi, \sigma \mid \lambda^{b,p} x : s.\, e\ v \longrightarrow \pi, \sigma \mid e\{v/x\}$.

From $\Sigma \mid \Gamma; a \vdash \lambda^{b,p} x : s.\, e\ v : t$ we have $\Sigma \mid \Gamma; a \vdash v : s$ and $\Sigma \mid \Gamma; a \vdash \lambda^{b,p} x : s.\, e : s \rightarrow^{b,p} t$ 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 10 and $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ from premise.

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

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

$$\Sigma \mid \Gamma; a \vdash v : t' \tag{7}$$

and $\Gamma \vdash m$ by (T-Ref), and $(\Sigma, r : t') \mid \Gamma; a \vdash v : t'$ by store typing Lemma 13, 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 $\pi, \sigma$ is well-typed and the assumption that $r \notin \mathit{dom}(\sigma)$) hence $(\Sigma, r : t') \mid \Gamma; a \vdash \pi, (\sigma, r \mapsto v)$ by (7) and store extension (Lemma 12), which completes the second part of the needed result.

**Case** $\pi, \sigma \mid\, !r \longrightarrow \pi, \sigma \mid v \;\text{ if }\; \sigma(r) = v$.

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 $\pi, \sigma$ is well typed and Definition 2. 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** $\pi, \sigma \mid r := v \longrightarrow \pi, \sigma[r \mapsto v] \mid ()$.

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

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

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 (8), premise that the store $\pi, \sigma$ is well typed, and the store update Lemma 11, which completes the second part of the needed result.

**Case** $\pi, \sigma \mid \mathcal{E}[\,\mathtt{fork}\; e\,] \longrightarrow \pi, \sigma \mid \mathcal{E}[\,()\,], e$.

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

$$\begin{aligned} \Sigma' \mid \Gamma'; a &\;\vdash\; e : \mathtt{Unit} && (9)\\ \Sigma' \mid \Gamma'; a &\;\vdash\; \mathtt{fork}\; e : \mathtt{Unit} && (10) \end{aligned}$$

for some $\Sigma'$ and $\Gamma'$ by (T-Fork). From $\Sigma \mid \Gamma; a \vdash \mathcal{E}[\,\mathtt{fork}\; e\,] : t$ and (10) we have $\Sigma \mid \Gamma; a \vdash \mathcal{E}[\,()\,] : t$ by (T-Unit) and replacement Lemma 9. From the latter and (9) 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** $\pi, \sigma \mid f_i, f'_j \longrightarrow \pi, \sigma \mid f_i \;\text{ if }\; i < j$.

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** $\pi, \sigma \mid \mathcal{E}[\,\texttt{isolated}\ \bar{l}\ e\,] \longrightarrow \pi, \sigma \mid \mathcal{E}[\,()\,], \texttt{task}\ pv\ e$.

From $\Sigma \mid \Gamma; a \vdash \mathcal{E}[\,\texttt{isolated}\ \bar{l}\ e\,] : t$, by (T-Isol) we have $\Sigma' \mid \Gamma'; a \vdash l_i : o_{l\,i}$ for all $i = 1..|\bar{l}|$, and $\Sigma' \mid \Gamma'; \{o_{l\,1}\} \cup ... \cup \{o_{l\,|\bar{l}|}\} \vdash e : t'$ for some $t'$, and $\Sigma' \mid \Gamma'; a \vdash \texttt{isolated}\ \bar{l}\ e : \texttt{Unit}$ for some $\Sigma'$ and $\Gamma'$. Hence $\Sigma \mid \Gamma; a \vdash \mathcal{E}[\,()\,] : t$ by (T-Unit) and replacement Lemma 9. Since $\Sigma' \mid \Gamma'; a \vdash \texttt{task}\ pv\ e : \texttt{Unit}$ by (T-Task), hence $\pi, \sigma \mid \mathcal{E}[\,()\,], \texttt{task}\ pv\ e : t$ 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** $\pi, \sigma \mid \texttt{task}\ pv\ v \longrightarrow \pi, \sigma \mid ()$.

From $\Sigma \mid \Gamma; a \vdash \texttt{task}\ pv\ v : t$ we have $t = \texttt{Unit}$ by (T-Task), 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** $\pi, \sigma \mid \texttt{newlock}\ x\!:\!m\ \texttt{in}\ e \longrightarrow (\pi, l \mapsto 0), \sigma \mid e\{l/x\}\{o_l/m\}$ if $l \notin \mathit{dom}(\pi)$.

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

$$(\Sigma, l : \{0,1\}, o_l :: \texttt{Lock}) \mid (\Gamma, m :: \texttt{Lock}, x : m); a \vdash e : t \tag{11}$$

by store typing (Lemma 13). 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 (11), substitution (Lemma 10) and the definition of a singleton lock 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 $\pi, \sigma$ 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 12).

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

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

$$\Sigma \mid \Gamma; a \ \vdash\ l : o_l \quad o_l \in a \tag{12}$$

$$\Sigma \mid \Gamma; a \ \vdash\ e : t \tag{13}$$

by (T-Sync). From (12) and $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (premise), we have

$$\Sigma(l) = \{0,1\} \tag{14}$$

and $\Sigma(o_l) = \texttt{Lock}$ by (T-LockLoc). From (12) and (13) 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 (14) 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 11, which completes the second part of the needed result.

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

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

$$\Sigma \mid \Gamma; a \quad \vdash \quad l : o_l \tag{15}$$

$$\Sigma \mid \Gamma; a \quad \vdash \quad v : t \tag{16}$$

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 (15), 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 11, 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 $\Sigma'$, $\Gamma'$ and $t'$ such that $\Sigma' \mid \Gamma' \vdash e : t'$;
- if there are no $\Sigma'$, $\Gamma'$ and $t'$ such that $\Sigma' \mid \Gamma' \vdash e : t'$, then there are no $\Sigma$, $\Gamma$, 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 7 (Permutation)** If $\Sigma \mid \Gamma; a \vdash T : t$ and $\Delta$ is a permutation of $\Gamma$, 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 $\Gamma$ of a judgment $\Gamma \vdash e : t$ that are not free in the expression $e$ may be ignored.

**Lemma 8 (Weakening)** If $\Gamma(x) = \Gamma'(x)$ for all $x \in \mathit{fv}(e)$ 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 9 (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 [38] (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 10 (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 $\Sigma$, 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 $t' = t_1 \rightarrow^{b,p} t_2$ 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 8) 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 $\Gamma \vdash \lambda^{b,p} x' : t_1 .\, e_1\{v/x\} : t_1 \rightarrow^{b,p} t_2$. But this is precisely the needed result, since, by the definition of substitution, $\Gamma \vdash (\lambda^{b,p} x' : t_1 .\, e_1)\{v/x\} : t_1 \rightarrow^{b,p} t_2$.

**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 $(\Gamma, x : t); a \vdash e_1 : t_1 \rightarrow^{b,p} t'$ for some $t_1$ and $b \subseteq a$, and

$$\Gamma; a \vdash e_1\{v/x\} : t_1 \rightarrow^{b,p} t' \tag{17}$$

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

$$\Gamma; a \vdash e_2\{v/x\} : t_1 \tag{18}$$

by induction hypothesis.

Then by (T-App) with (17) and (18) $\Gamma; a \vdash e_1\{v/x\}\ e_2\{v/x\} : t'$. But this is precisely the needed result, since, by the definition of substitution $\Gamma; a \vdash (e_1\ e_2)\{v/x\} : t'$.

**Case** $e = \texttt{ref}_m\, e \colon \texttt{Ref}_m\, t_1$ .

By (T-Ref), it follows from the assumption $(\Gamma, x : t) \vdash \texttt{ref}_m\, e : t'$ that $t' = \texttt{Ref}_m\, t_1$, 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 $\Gamma \vdash \texttt{ref}_m\, e\{v/x\} \colon \texttt{Ref}_m\, t_1$ . But this is precisely the needed result, since, by the definition of substitution $\Gamma \vdash (\texttt{ref}_m\, e)\{v/x\} \colon \texttt{Ref}_m\, t_1$ .

**Case** $e = !e$.

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

$$\Gamma; a \vdash e_1\{v/x\} : \texttt{Ref}_m\, t_1 \tag{19}$$

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

$$\Gamma; a \vdash e_2\{v/x\} : t_1 \tag{20}$$

by induction hypothesis.

Then by (T-Assign) with (19) and (20) $\Gamma; a \vdash e_1\{v/x\} := e_2\{v/x\} : \texttt{Unit}$. 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

$$m \in a \quad . \tag{21}$$

By induction hypothesis

$$\Gamma; a \vdash e_1\{v/x\} : m \quad . \tag{22}$$

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

$$\Gamma; a; p \cup \{m\} \vdash e_2\{v/x\} : t' \quad . \tag{23}$$

Then by (T-Sync) with (21), (22) and (23) $\Gamma; a; p \vdash \texttt{sync}\ e_1\{v/x\}\ e_2\{v/x\} : t'$. 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** $e = \texttt{isolated}\ e_1, ..., e_n\ e_0$.

From $(\Gamma, x : t); a; p \vdash \texttt{isolated}\ e_1, ..., e_n\ e_0 : t'$ 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

$$\Gamma; a; p \vdash e_i\{v/x\} : m_i \quad \text{for all} \;\; i = 1..n \tag{24}$$

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

$$\Gamma; \{m_1\} \cup ... \cup \{m_n\}; \emptyset \vdash e_0\{v/x\} : t_0 \tag{25}$$

by induction hypothesis.

Then by (T-Isol) with (24) and (25) $\Gamma; a; p \vdash \texttt{isolated}\ e_1\{v/x\}, ..., e_n\{v/x\}\ e_0\{v/x\} : \texttt{Unit}$. But this is precisely the needed result, since, by the definition of substitution $\Gamma; a; p \vdash (\texttt{isolated}\ e_1, ..., e_n\ e_0)\{v/x\} : \texttt{Unit}$.

**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** $e = \texttt{task}\ pv\ f$.

By (T-Task), it follows from the assumption $(\Gamma, x : t); a \vdash \texttt{task}\ pv\ f : t'$ where $t' = \texttt{Unit}$, that $a = \{o_{l_1}, ..., o_{l_n}\}$ and $(\Gamma, x : t); a \vdash l_i : o_{l_i}$ and $(\Gamma, x : t); a \vdash pv(l_i) : \mathbf{Nat}$ for all $i = 1..n$, and

$$\Gamma; a \vdash pv(l_i)\{v/x\} : \mathbf{Nat} \quad \text{for all} \;\; i = 1..n \tag{26}$$

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

$$\Gamma; a \vdash f\{v/x\} : t \tag{27}$$

by induction hypothesis.

Then by (T-Task) with (26, 27) $\Gamma; a \vdash \texttt{task}\ pv\{v/x\}\ f\{v/x\} : \texttt{Unit}$. But this is precisely the needed result, since, by the definition of substitution $\Gamma; a \vdash (\texttt{task}\ pv\ f)\{v/x\} : \texttt{Unit}$. $\Box$

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 lock location, or $\pi, \sigma[l \mapsto v]$ if $l$ is a reference cell location. See the canonical forms Lemma 14 in §A.3.2 that states the possible shapes of values of various types.

**Lemma 11 (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 2). $\square$

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 lock location, or $\pi, (\sigma, l \mapsto v)$ if $l$ is a reference cell location. See the canonical forms Lemma 14 in §A.3.2 that states the possible shapes of values of various types.

**Lemma 12 (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 $\Sigma \mid \Gamma; a \vdash \pi, \sigma$. $\square$

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 13 (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. $\square$

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

PROOF. If $\pi, \sigma \mid T \longrightarrow (\pi, \sigma)' \mid T'$, then $T = \mathcal{E}[\, e_1 \,]$ and $T' = \mathcal{E}[\, e_2 \,]$, and $\pi, \sigma \mid e_1 \longrightarrow (\pi, \sigma)' \mid e_2$ 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 9. Then the result follows by induction on the length of the reduction sequence $\pi, \sigma \mid T \longrightarrow^* (\pi, \sigma)' \mid T'$. $\square$

### A.3.2 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 14 (Canonical Forms)**
1) If $v$ is a value of type `Unit`, then $v$ is $()$.
2) If $v$ is a value of type $t \rightarrow^{a,p} s$, then $v = \lambda^{a,p} x : t.\, e$.
3) If $v$ is a value of type $m$, then $v$ is a lock location.
5) If $v$ is a value of type $\mathtt{Ref}_m\, t$, 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 2 and the extended grammar in the upper part of Figure 3. $\Box$

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 $\pi$, if there is some evaluation context $\mathcal{E}[\,\mathtt{insync}\; l\, e\,]$ in the thread sequence for any lock location $l$, then $\pi(l)$ should contain 1, marking that the lock has been acquired. As for the store $\sigma$, containing the content of each reference cell, we may only require that it is well typed.

**Definition 3** Suppose $\pi, \sigma$ 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 $f_i = \mathcal{E}[\,\mathtt{insync}\; l\, e\,]$ for some $l$, there is $\pi(l) = 1$.

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

According to Lemma 15, the property $\vdash_{wf} \pi, \sigma \mid \overline{f}$ is maintained during evaluation.

**Lemma 15 (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 $\pi, \sigma$. Suppose that $e_0 = \mathcal{E}[\,\mathtt{sync}\; l\; e\,]$ for some context $\mathcal{E}$, and $\pi(l) = 0$. (Note that when a lock 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 $e_0 = \mathcal{E}'[\,\mathtt{insync}\; l\, e'\,]$ 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 $e_1 = \mathcal{E}[\,\mathtt{insync}\; l\, e\,]$. But then, after reduction step, we have $\pi(l) = 1$ (again by (R-Sync)). Moreover, by type preservation Theorem 4, 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. $\Box$

A state $\pi, \sigma \mid T$ is *deadlocked* if there exist only evaluation contexts $\mathcal{E}$, such that $T = \mathcal{E}[\,\mathtt{sync}\; l\; e\,]$ 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 $\Sigma$). Then either $T$ is a value or else, for any store $\pi, \sigma$ 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 lock(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 $\vdash e_1 : t \rightarrow^{b,p} s$ 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 $T = e_1\ e_2$ . □

# B Dynamic Correctness of the BVA Algorithm

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

## B.1 Assumptions and Definitions

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 task 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 tasks 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 locks). 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 4** (Noninterference) 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 tasks, such that given any verlock, the order of acquiring the verlock by tasks in the concurrent run is the same as in $R^{s}$.

Below we explain each step of the algorithm, given by evaluation rules (BVA-0-3) in Figure 4. We require steps (BVA-1) and (BVA-2) to be atomic. We write $gv_l$ and $lv_l$ as shorthand for $gv(l)$ and $lv(l)$.

**(BVA-0)** Upon lock creation, by rule (R-Lock), initialize global and local counters of the new lock to zero.

**(BVA-1)** At the moment of spawning a new task $k$ using `isolated` $\bar{l}$ $e$, by rules (R-Isol), for each lock $l_i$ where $i = 1, .., |\bar{l}|$, that may be requested by this task, increase counter $gv_{l_i}$ by one. Create a fresh (read-only) map $pv_k$ that contains bindings from the locks $l_i$ to their upgraded versions $gv_{l_i}$.

**(BVA-2)** A task $k$ can acquire a lock $l$ only when, by rule (R-Sync), the lock is free and the task holds a (private) version of this lock that – when downgraded by one – matches the current (local) version maintained by the lock, i.e.

$$pv_k(l) - 1 = lv_l \quad . \tag{28}$$

**(BVA-3)** After a task $k$ has completed its execution, i.e. all threads of the task have terminated, by rule (R-Task), for each lock $l_i$, where $i = 1, .., |\bar{l}|$, wait until condition (28) is true, then upgrade a local version of each lock $l_i$, so that $lv_{l_i} = pv_k(l_i)$.

Essentially, the BVA algorithm implements ordering of lock 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 task-free state that is not obtainable by some serialized evaluation of tasks. Note that we do not require a program to terminate. However, we consider its correctness only for a set of tasks that will eventually terminate.

To prove the correctness of the algorithm, we only need to show that all tasks 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 task `task` $pv$ $e$ where $pv(l)$ is defined, we define *access* of this task 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 `task` $pv$ $e$ to a lock $l$ is *defined* if $pv(l)$ is defined.

Access $a_k = (pv_k(l), lv_l)$ of a task $k$ is *valid* if condition (28) is true. A task *gets* a valid access $(pv_k(l), lv_l)$ when condition (28) is becoming true.

## B.2 Verlock Access

**Lemma 16 (Verlock Safety)** A verlock can be acquired only by a task which has valid access to the verlock.

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

**Lemma 17 (Access Liveness)** Each access of a given task in a concurrent run will be eventually valid, provided that all tasks terminate.

Proof. Let $k_0$ be the first task, 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 (28) is true. Consider a task $k_1$ 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 (28) is false $(2 - 1 \neq 0)$. However, since we assumed that tasks 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 tasks, we will get the needed result. □

**Lemma 18 (Verlock Liveness)** Each non-free verlock requested by a task will be eventually acquired, provided that it will be released.

Proof. Straightforward from access liveness Lemma 17 and the premise of (R-Sync). □

**Lemma 19 (Private-Version Uniqueness)** Each task has a unique private version of each verlock during task 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 20 (Access Uniqueness)** For each verlock and any task 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 19. □

**Lemma 21 (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 tasks at any time, from private-version uniqueness Lemma 19, we have that at any given time, there is only one task which can have access for which validity condition (28) is true. Hence, we obtain the needed result. □

**Lemma 22 (Access Privacy)** A valid access $a_k$ of a task $k$ can be invalidated only by task $k$.

Proof. Consider a valid access $a_k = (pv_k(l), lv_l)$ of some task $k$ to a verlock $l$. By access uniqueness Lemma 20, there is no other task $k'$ 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 21, we know that it is not possible that some other task 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 task $k$ can only upgrade $lv_l$ if (28) 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 23 (Valid-Access Preservation)** If a task 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 21 and access privacy Lemma 22. □

**Lemma 24 (Verlock-Set Mutual Exclusion)** As long as a task is allowed to acquire a verlock $l$, no other task can acquire verlock $l$.

Proof. Straightforward from valid-access-preservation Lemma 23 and verlock safety Lemma 16. □

By verlock-set mutual exclusion Lemma 24, and the fact that we are not interested in the relative order of lock acquisitions made by the same task (since *any* such order would satisfy Definition 4 of noninterference), we can represent all acquisitions of a given verlock made by a given task 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 task at most once. By Lemma 24, the proven result will be valid for any system.

## B.3 Access Ordering

**Lemma 25 (Access Ordering)** The order of acquiring a verlock by tasks corresponds to the order in which tasks got valid access to it.

Proof. Immediate by verlock safety Lemma 16 and verlock-set mutual exclusion Lemma 24. □

**Lemma 26 (Valid-Access Ordering)** The relative order of getting valid access to a verlock by tasks corresponds to the order of creating the tasks.

Proof. Consider a task $k$, which gets valid access to some lock $l$. Access becomes valid when condition (28) becomes true. By step (BVA-3), this occurs when some other task $k'$ upgrades a local version $lv_l$ by 1. By access privacy and valid-access mutual exclusion, the task $k'$ 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 (28), we can derive that

$$pv_{k'}(l) = pv_k(l) - 1 \; . \tag{29}$$

Moreover, from step (BVA-1), we know that the order of private versions corresponds to the order of creating tasks, i.e. if $k_i$ has been created before $k_j$, then $pv_{k_i}(l) < pv_{k_j}(l)$ for each lock $l$ such that both tasks have defined access to it. Hence, from (29), we know that $k'$ has been created before $k$. Finally, by induction on tasks we obtain the needed result. □

**Lemma 27 (Total Ordering)** The relative order of acquiring a verlock by tasks is the same for every verlock.

Proof. Immediate from verlock safety Lemma 16, verlock-set mutual exclusion Lemma 24, and access ordering Lemma 25, valid-access ordering Lemma 26, and the fact that the order of creating tasks is total (by step (BVA-1)). □

**Lemma 28 (Natural Ordering)** The order of acquiring verlocks by tasks in a concurrent run is the same as in some serial run.

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

From verlock safety Lemma 16, valid-access ordering Lemma 26, verlock-set mutual exclusion Lemma 24, and total ordering Lemma 27, it is straightforward that any concurrent run has the "natural order" property. Moreover, since we only consider isolation for expressions that reached a task-free state (see Lemma 6), hence we are allowed to consider only concurrent runs in which all tasks 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 18, 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. □

## B.4 Isolated Execution

We conclude that the BVA algorithm can be used to implement the isolated execution of tasks.

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

Proof. By natural ordering Lemma 28, the noninterference property is satisfied in any concurrent run in which verlocks are acquired when permitted by the algorithm, which completes the proof. □

## References


[1] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. In *Proceedings of ECOOP 2002: The 16th European Conference on Object-Oriented Programming*, volume 2374 of *LNCS*. Springer, June 2002.

[2] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[3] A. Bestavros. Advances in real-time database systems research. *ACM SIGMOD Record*, 25(1):3–8, Mar. 1996.

[4] A. P. Black, V. Cremet, R. Guerraoui, and M. Odersky. An equational theory for transactions. In *Proceedings of FSTTCS '03: The 23rd International Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 2914 of *LNCS*. Springer, Dec. 2003.

[5] P. K. Chrysanthis and K. 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.

[6] D. Detlefs, M. Herlihy, and J. Wing. Inheritance of synchronization and recovery properties in Avalon/C++. *IEEE Computer*, 21(12):57–69, Dec. 1988.

[7] E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965.

[8] J. L. Eppinger, L. B. Mummert, and A. Z. Spector, editors. *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kaufmann, 1991.

[9] C. Flanagan and M. Abadi. Types for safe locking. In *Proceedings of ESOP 1999: The 8th European Symposium on Programming*, volume 1576 of *LNCS*. Springer, Mar. 1999.

[10] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of PLDI 2003: The ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2003.

[11] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proceedings of CONCUR 1996: The 7th International Conference on Concurrency Theory*, volume 1119 of *LNCS*. Springer, Aug. 1996.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison Wesley, 2000.

[13] M. H. Graham. Issues in real-time data management. Technical Report SEI-TR-17, Software Engineering Institute, Carnegie-Mellon University, July 1991.

[14] N. Haines, D. Kindred, J. G. Morrisett, S. M. Nettles, and J. M. Wing. Composing first-class transactions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1719–1736, Nov. 1994.

[15] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of OOPSLA 2003: The 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2003.

[16] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[17] 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, 1972.

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

[19] S. Jagannathan and J. Vitek. Optimistic concurrency semantics for transactions in coordination languages. In *Proceedings of Coordination 2004: The 6th International Conference on Coordination Models and Languages*, volume 2949 of *LNCS*. Springer, Feb. 2004.

[20] S. P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proceedings of POPL 1996: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 1996.

[21] K.-J. Lin and C.-S. Peng. Enhancing external consistency in real-time transactions. *ACM SIGMOD Record*, 25(1):26–28, 1996.

[22] S. Mena, A. Schiper, and P. T. Wojciechowski. A step towards a new generation of group communication systems. In *Proceedings of Middleware 2003: The 4th ACM/IFIP/USENIX Middleware Conference*, volume 2672 of *LNCS*. Springer, June 2003.

[23] Microsoft. *MTS*. http://www.microsoft.com/.

[24] P. Panangaden and J. Reppy. The Essence of Concurrent ML. In *ML with Concurrency: Design, Analysis, Implementation, and Application.*, Monographs in Computer Science, pages 5–29. Springer, 1997.

[25] G. Parrington and S. Shrivastava. Implementing concurrency control in reliable distributed object-oriented systems. In *Proceedings of ECOOP 1988: The 2nd European Conference on Object-Oriented Programming*, volume 322 of *LNCS*. Springer, Aug. 1988.

[26] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[27] G. D. Plotkin. Call-by-name, call-by-value and the $\lambda$-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[28] R. Rajwar and J. R. Goodman. Transactional execution: Toward reliable, high-performance multithreading. *IEEE Micro*, 23(6), Nov./Dec. 2003.

[29] *The SAMOA Protocol Framework*. http://lsrwww.epfl.ch/samoa.

[30] P. Sewell, P. T. Wojciechowski, and B. C. Pierce. Location-independent communication for mobile agents: a two-level architecture. In *Internet Programming Languages*, volume 1686 of *LNCS*, pages 1–31. Springer, 1999.

[31] L. Shu and M. Young. Correctness criteria and concurrency control for real-time systems: a survey. Technical Report SERC-TR-131-P, Purdue University, Nov. 1992.

[32] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts, Sixth Edition*. John Wiley & Sons, 2002.

[33] Sun. *EJB*. http://java.sun.com/.

[34] J. Vitek, S. Jagannathan, A. Welc, and A. L. Hosking. A semantic framework for designer transactions. In *Proceedings of ESOP 2004: The 13th European Symposium on Programming*, volume 2986 of *LNCS*. Springer, March/April 2004.

[35] G. Weikum and G. Vossen. *Transactional Information Systems*. Morgan Kaufmann, 2002.

[36] J. M. Wing, M. Faehndrich, J. G. Morrisett, and S. Nettles. Extensions to Standard ML to support transactions. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, 1992.

[37] P. Wojciechowski, O. Rütti, and A. Schiper. SAMOA: A framework for a synchronisation-augmented microprotocol approach. In *Proceedings of IPDPS 2004: The 18th IEEE Parallel and Distributed Processing Symposium*, Apr. 2004.

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