# Isolation-only Transactions by Typing and Versioning

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

## ABSTRACT

In this paper we design a language and runtime support for isolation-only, multithreaded transactions (called tasks). Tasks allow isolation to be *declared* instead of having to be encoded using the low-level synchronization constructs. 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.

**Categories and Subject Descriptors:** D.3.3 [Programming Languages]: Language Constructs and Features—*Abstract data types, Concurrent programming structures*

**General Terms:** Design, Languages, Reliability, Theory, Verification.

**Keywords:** programming languages, concurrency, type theory, transactions, isolation, declarative synchronization, abstract types, singleton kinds, lambda calculus.

## 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* [32, 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 [7, 8, 33, 14, 22, 30] 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 performing input/output (I/O). 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 networked applications, which are inherently concurrent. A distinct feature of these applications is that they perform many I/O operations such as sending and receiving network messages; 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 their development. Unfortunately, traditional transaction techniques can seldom be transferred from the database to time-critical domain without change; the performance considerations are too different [29, 13, 3].

This paper describes a language and runtime support for isolation-only, local transactions, called *tasks*. The *isolation* property [32] (also known as *serializability*) ensures that the concurrent execution of tasks is equivalent to an execution in which the tasks are serialized. Tasks allow isolation to be *declared* instead of having to be encoded using the low-level synchronization means.

Contrary to similar constructs for declaring atomic blocks that we describe in §6, our tasks can perform arbitrary operations, including I/O, with the isolation guarantee. The main idea is to avoid the need for rollback at runtime (due to e.g., conflicts on task operations), by tightly controlling the order of such operations and guaranteeing that once started a task cannot run into conflicts. No explicit rollback by the program is allowed.

The isolation guarantee in our language stems from three sources:

- compile time enforcement that each shared data location (or an I/O operation) is protected by a lock and that threads acquire the corresponding lock before accessing the location (or performing the I/O operation); this is based on previous work of Abadi and Flanagan on types for *safe locking* [9],
- compile time enforcement that requires that all locks to be acquired during a task are declared at the beginning of the task,

*PPDP'05,* July 11–13, 2005, Lisbon, Portugal.

- a runtime locking strategy that assigns versions to threads that allow them to acquire locks so that isolation is preserved.

Our approach allows multiple threads within an isolated task. These threads share memory and are not isolated from each other. We describe an implementation of an example networked application that uses multithreaded tasks in §5.

## 1.1 Design Choices

We can identify several requirements of time-critical applications that also apply to networked applications (based on survey papers [29, 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 replication have been developed to reduce the possibility of failure; they are however beyond scope of this paper.

*2. No rollback is allowed.* Contrary to databases, task operations may not be recoverable. For instance, let us consider a multi-task middleware. Tasks on each machine are executed locally but they may exchange messages over the network with other tasks; the messages are then delivered to any distributed applications built on top of the middleware. Full-scale recovery of tasks in such cases is usually too expensive (it requires some form of distributed agreement [32, 1] between tasks) and impractical (since it would also require the applications to be able to rollback their state).

*3. Serializability is sometimes too conservative.* Several authors described the limitations of serializability as a correctness criterion (see articles, e.g. [20, 29], for details).

Based on the above requirements, we now motivate and introduce the main features of our language.

**Fine-grain, rollback-free concurrency control** In our previous work [34], we have introduced (informally) several novel pessimistic concurrency control algorithms for scheduling critical task operations with the isolation property. The algorithms are rollback-free. In §5 we describe an implementation of our language that uses these algorithms.

Roughly, a greater degree of concurrency leads to higher performance. The degree depends on the amount of information available to the concurrency controller. Knowing more information about intended program behaviour, such as predefined patterns of acquiring locks and semantic information about the meaning of data and operation performed, allows one to select an algorithm that permit more concurrency.

In this paper we give a rigorous design of our language, and describe formally the semantics of its constructs using a *Basic Versioning Algorithm* (BVA). It permits less concurrency than other, more complex algorithms that we describe in [34], but it is more convenient to illustrate a novel *hybrid approach* that combines concurrency control with typing.

**Typing for safe concurrency control** Our language has a construct `isolated` $\overline{x}$ $e$, that spawns an isolated task $e$, where $\overline{x}$ are the data declared for the concurrency controller. To make the language safe, we propose in this paper a type system that can verify if $\overline{x}$ is correct; to our knowledge it is the first presentation of a type system for such application.

The type system builds on Flanagan and Abadi's [9] type system for detection of race conditions. We have used their solution to ensure that all accesses to shared data are protected by locks. This guarantee could be relaxed in the future by refining the type system, e.g. objects known to be immutable need not be visible to the concurrency controller when accessed, and so they could be left unprotected.

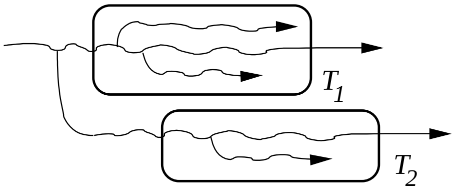

**Figure 1: Concurrent, multithreaded transactions**

**Isolation of arbitrary operations (including I/O)** Our language allows *arbitrary* sets of operations to be isolated, including I/O operations such as message output and input. These operations just need to be protected by *versioning locks* (or *verlocks*). Verlocks are similar to locks. They however extend the standard locking principle with a runtime locking strategy for isolation preservation. The programmer can therefore use verlocks to declare operations that must be serialized. Operations that are not protected by verlocks are not serialized; this design choice supports the requirement about relaxing isolation, e.g. message receipts in our example application in §5 are not serialized.

Alternatively, (ver)locking could be left as an implementation issue of the `isolated` construct by letting the compiler to place verlocks automatically. In this paper, however, we keep verlocks as a language construct, as it allows us to explain the semantics and typing rules at the level of detail that is required for rigorous proofs of isolation.

**Multithreaded tasks** To support cases when isolation is too restrictive, we allow tasks to be multithreaded. Threads are lightweight processes that can communicate using shared mutable data and synchronize (in scope of a task) by acquiring and releasing verlocks (with caution to avoid deadlock); other synchronization means such as monitors can also be used. Individual threads may fork and e.g. start other tasks.

Figure 1 illustrates two concurrent, multithreaded tasks $T_1$ and $T_2$. Execution of each task is atomic with respect to other tasks (run on the same machine). We do not require however threads within a task to be serializable; thus, they can engage in two-way communication using shared data. (Note that constructs such as *nested transactions* [32] do not apply here, as they normally do not relax isolation between subtransactions, and they depend on rollback-recovery.)

## 1.2 Contribution

We make several contributions:

- We present an *operational semantics* of tasks and verlocks; the semantics has been split into a dynamic semantics of the host language constructs, and of the concurrency controller; we have used the semantics to formalize and prove correct the BVA algorithm.

- We have shown several results and theorems about our *type-directed approach* to concurrency control of rollback free transactions. The main result is that well-typed programs satisfy the isolation property.

- To our best knowledge we give the first rigorous proof of isolation preservation and progress (up to deadlocks between threads of the same task) that makes data accesses explicit and deals with multiple threads within an atomic block.

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. §4 states the main results, including type soundness and dynamic correctness of the BVA algorithm. §5 sketches an implementation of tasks and an example networked application. §6 discusses related work and §7 concludes.

## 2. EXAMPLE

We use an example of transactions with irrevocable I/O effects from [29]. Consider a central air route surveillance station that controls air traffic in a large geographic area. It receives aircraft positions from local stations – one per geographic region $i$ – and records them in a corresponding "track table" tab$i$; in parallel, it outputs control data.

Below are two concurrent tasks $T_1$ and $T_2$, expressed using our language (and objects and some syntactic sugar).

```
newlock x : TabA in
newlock y : TabB in
isolated x,y   (* task T1: hand-over *)
(
  sync x tabA.withdraw(aircraft);
  sync y tabB.deposit(aircraft);
)
isolated x,y   (* task T2: control *)
(
  view_tabA := sync x tabA.get();
  view_tabB := sync y tabB.get();
  analyseAndOutput(view_tabA, view_tabB);
)
```

Task $T_1$ records aircraft movement based on information from 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 of an aircraft must not disappear or appear in more than one table. (For simplicity, we only encoded the hand-over from region $A$ to $B$.)

Suppose some aircraft moved from region $A$ to $B$ while task $T_1$ is updating the track tables. Meanwhile, the task $T_2$ analyses the traffic pattern in the controlled area and produces a warning if two aircraft fail to maintain minimum separation. For this, $T_2$ must obtain a snapshot view of the controlled area by reading the tables.

$T_2$ could obtain an inconsistent view if it first retrieves data in region $B$ before $T_1$ 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 $T_1$ and $T_2$ 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 a 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 (i.e. each task is executed by a new thread). 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. Tasks can perform I/O and spawn threads, e.g. task $T_2$ may output a warning message, and spawn a new thread for auditing.

Flanagan and Abadi's type system provides guarantees for the concurrency scheduler that all 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 our abort-free versioning algorithm.

Execution of tasks $T_1$ and $T_2$ 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.).

**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 $\mathtt{Ref}_m\, t$ 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 $\mathtt{ref}_m\, e$ (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 `let` $x = e_1$ `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

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

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**

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 §3.3.

## 3.2 Operational Semantics

We specify the operational semantics using the rules defined in Figure 3. 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 be 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. The evaluation terminates once all threads have been reduced to values, in which case the value $v_0$ of the initial, first thread $T_0$ is returned as the program's result (typing will ensure that other values are empty values). Subscripts in values reduced from threads denote the sequence number of the thread, i.e. $v_i$ is reduced from $i$'s thread, denoted $T_i$ $(i = 0, 1..)$. The execution of threads can be arbitrarily interleaved. Since different interleavings may produce different results, the evaluator $\mathit{eval}(e, v_0)$ is therefore a relation, not a partial function.

Below we describe reduction rules in Figure 3. The rules in the middle are common for all versioning concurrency control algorithms, while the rules in the bottom part of the figure describe our example algorithm.

The first four evaluation rules are the standard rules of a call-by-value $\lambda$-calculus [26], 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.

## 3.3 Basic Versioning Algorithm

Below we describe the *Basic Versioning Algorithm* (BVA) for "isolated evaluation" of tasks. For clarity, we have chosen one of the simplest algorithms possible. The semantics can be however easily extended to optimized algorithms of [34] that permit more concurrency.

The algorithm implements a runtime locking strategy that assigns essentially tickets to threads that allow them to acquire verlocks. The tickets are monotonically increasing counters, one per lock. On task entry, a thread obtains incremented ticket values (called *versions*) for all the verlocks that it wants to acquire during the task. It can then acquire these verlocks only when the corresponding verlocks service count has reached its ticket count. Since tickets for

**State Space**

$$\begin{array}{rcccl} S & \in & \mathit{State} & = & \mathit{LockStore} \times \mathit{RefStore} \times \mathit{ThreadSeq} \\ \pi & \in & \mathit{LockStore} & = & \mathit{LockLoc} \to \{0,1\} \\ \sigma & \in & \mathit{RefStore} & = & \mathit{RefLoc} \to \mathit{Val} \\ l & \in & \mathit{LockLoc} & \subset & \mathit{Var} \\ r & \in & \mathit{RefLoc} & \subset & \mathit{Var} \end{array}$$

$$\begin{array}{rcccl} \mathit{pv} & \in & \mathit{VerMap} & \subset & \mathit{LockLoc} \to \mathbf{Nat} \\ T & \in & \mathit{ThreadSeq} & ::= & f \mid T, T \\ f & \in & \mathit{Exp}_{\mathit{ext}} & ::= & x \mid v \mid f\ e \mid v\ f \mid \texttt{ref}_m\, f \mid\ !f \\ \multicolumn{5}{l}{\mid f := e \mid r := f \mid \texttt{newlock}\ x\!:\!m\ \texttt{in}\ e \mid \texttt{sync}\ f\ e \mid \texttt{insync}\ l\, f} \\ \multicolumn{5}{l}{\mid \texttt{fork}\ e \mid \texttt{isolated}\ \overline{f}\overline{e}\ e \mid \texttt{isolated}\ \overline{l}\overline{f}\ e \mid \texttt{task}\ \mathit{pv}\ T} \end{array}$$

**Evaluation Contexts**

$$\begin{array}{rcl} \mathcal{E} & = & [\,] \mid \mathcal{E}\ e \mid v\ \mathcal{E} \mid \texttt{ref}_m\, \mathcal{E} \mid\ !\mathcal{E} \mid \mathcal{E} := e \mid r := \mathcal{E} \mid \texttt{sync}\ \mathcal{E}\ e \mid \texttt{insync}\ l\, \mathcal{E} \\ & & \mid \texttt{isolated}\ \bar{l}\mathcal{E}\bar{e}\ e \mid \texttt{task}\ \mathit{pv}\ \mathcal{E} \mid \mathcal{E}, T \mid T, \mathcal{E} \end{array}$$

**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**

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

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

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

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

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

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

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

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

**Transition Rules of Basic Versioning Algorithm (BVA)**

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

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

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

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

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

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

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

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

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

all verlocks are obtained atomically, this guarantees that tasks with conflicts (shared verlock) will commit in global order of task starts.

We define the algorithm formally via four operational semantics rules (BVA-0–3) in the lower part of Figure 3. The rules define creation and destruction of tasks, and verlock acquisition and release. Below we explain these rules.

**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 a natural number 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 (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 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. A task `task` $pv$ $e$ has *completed* or *terminated* if expression $e$ yields a value; see (R-Task). Then the task upgrades local counters of its verlocks and reduces to an empty value. (In [34] we describe variants of BVA that permit more concurrency by making the upgrades *during* task execution.) To ensure that the order of upgrades by all tasks is correct, the task completion is guarded by the condition that $pv(l) - 1$ must be equal $lv(l)$ for all $l$ in $dom(pv)$.

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*. The result states subsume data stored in all reference cells.

**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.)

Isolation has been proposed as the correctness condition of concurrency control algorithms [2]. It means, intuitively, that if the effects of one task are visible to some other task executing concurrently, then the opposite is not true, where an effect is usually defined as any change to the content of reference cells; from the perspective of a task, it appears that tasks execute sequentially rather than in parallel.

We extend the above definition of an effect, and assume that both assignment and dereference has an effect, respectively an *output* and *input* effect. Our definition of isolation is therefore more conservative; it 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.

**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's thread being blocked (any other threads are not blocked).

However, each lock will be eventually acquired (evaluation progress) if only tasks are themselves deadlock-free and terminate. We discuss the deadlock issue in §4.3, after explaining typing.

**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 §3.4.

## 3.4 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 4. 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. 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 4 has 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)).

**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**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**Figure 4: The first-order type system for the *iso*-calculus**

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 $Exp_{ext}$, and then to states as shown in Figure 5. 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 is the argument of the construct).

The first property is called *absence of race conditions* and is guaranteed by Abadi and Flanagan's type system for avoiding race conditions that we have extended. The second property is called *absence of non-declared verlocks* and is guaranteed by our extension of their type system. Based on the two properties of the type system, we prove that evaluation of well-typed, terminating programs satisfies the isolation property; the proof is in the technical report [35].

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 4, 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

**Judgments**

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

**Rules**

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

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

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

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

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

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

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

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

**Figure 5: Additional judgments and rules for typing states**

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 $\texttt{task}\; pv\; T$ 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 6). 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 $\texttt{task}\; pv\; T$ 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.

## 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

**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 6: Judgments and rules for reasoning about critical sections and tasks**

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.

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.

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

The only deadlocks possible in our language stem from either two threads of the same task that try to acquire two locks $l_1$ and $l_2$ in parallel but in a different order, or when a thread tries to acquire a lock again before releasing it. This means however that other tasks that want to acquire these locks will be also blocked. 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].

## 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 in [35] 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.

**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 Figure 3 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. 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. IMPLEMENTATION EXPERIENCE

**Protocol framework** We develop SAMOA [34] – a protocol framework that allows networked applications to be built from components that communicate using the framework's interface; an implementation as a Java [12] package is available [27]. The framework provides event communication, message flow control, and an `isolated` construct for spawning isolated tasks.

The programmer can choose among several versioning algorithms for rollback-free task execution, including the BVA algorithm, and also its two optimized variants that permit more parallelism by upgrading local version counters as soon as possible [34]. They however demand some additional data. For instance, one algorithm requires the least upper bound on the number of times a critical operation can be performed by a task, another one requires a graph (or pattern) that represents an order of possible critical operations. These data must be declared and passed as the argument of the `isolated` construct.

Implementing the type system described in §3.4 would make programming safe when choosing the BVA algorithm. In the future, we would like to extend the type system for the other algorithms. However, it may not be possible to verify some class of programs, e.g. supremum required by one algorithm cannot be derived if the program uses recursion.

**Example application** SAMOA has been used to implement modular *group communication* protocols [21, 34]. Our protocols execute some actions concurrently, e.g.: (*i*) for better response time when performing slow I/O operations, (*ii*) to avoid blocking while processing different types of messages, or (*iii*) to gain benefit of the multi-CPU architectures. In practice, messages of certain types that are received from the network (or application) spawn a new task. Any concurrent jobs in the context of the same message are performed by multiple threads within the task. Task isolation ensures however that each concurrent message is processed by the protocol using a consistent set of data, which made programming easier and less error-prone.

## 6. RELATED WORK

There have been recently many proposals of concurrent languages with novel synchronization primitives, e.g. the join-calculus language [11], Concurrent Haskell [24], Concurrent ML [23], Pict [25] and Nomadic Pict [28]. They enable to express complex synchronization code more easily than when using standard constructs, such as monitors and locks. This work is however orthogonal to the goals of our paper. We are primarily focused on high-level language support that provides automatic concurrency control.

The work in this paper builds on research in three areas: atomic transactions, language support for atomic blocks, and concurrency control algorithms. Below we discuss example work in these areas, and also on formalization.

**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 transactional platforms, such as Sun Enterprise JavaBeans (EJB) [30] and Microsoft Transaction Server (MTS) [22], and programming languages, such as Avalon/C++ [7] and Venari/ML [14, 33].

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, we intended to address the issue of local concurrency control in network protocols, rather than distributed transactions; the design considerations were therefore different. Contrary to traditional database transactions, our tasks never rollback their execution – we guarantee that I/O operations are performed *exactly once*, unless the process running all local tasks crashes.

Alternative approaches such as *compensations* [5], i.e. implicit or programmable procedures that can undo the effects of a transaction that fails to complete, do not apply here. Some I/O operations performed by tasks cannot be easily (or routinely) undone or compensated. For instance, we usually assume that an output of a network message either succeeds, i.e. the message is sent, or not, i.e. the message is not sent due to e.g., a socket error. The protocol designer should not be concerned with another case, when the message has been sent, but the operation needs some compensation due to conflicts on task operations.

**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. Below is the previous work closest to our own.

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 [17] 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.

Our decision to allow tasks to be multithreaded 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.

Moreover, applications that we consider may demand different levels of performance, isolation and real-time constraints; these varying demands will lead to a multiplicity of runtime concurrency controllers, based on a variety of scheduling algorithms (e.g. real-time algorithms [13]). Our intend is to allow the programmer to choose between different dynamic locking strategies, based on the available static information. Our declarative approach therefore differs from the above type-based approach to *verify* atomicity.

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) [18]. 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* (STM) which groups together series of memory accesses and makes them appear atomic.

The main difference between their approach and ours is the lack of a need for rollback. Unlike our pessimistic concurrency control, their implementation of atomicity depends on rollback and recovery. This restricts the availability of I/O operations within an atomic block. For instance, the STM-based implementation of atomic blocks in Haskell [16] forbids all operations that may have irrevocable I/O effects, which limits the scope of possible applications.

A plausible option could be based on buffering input operations (for possible recovery) and flushing all output operations on transaction commit (to prevent their duplication due to rollback). However, it does not seem to support an arbitrary pattern of I/O communication at real time.

**Concurrency control** Our versioning concurrency control algorithms have some resemblance with *two-phase locking* [2, 32]. However, instead of acquiring all locks needed (in the 1st phase) and releasing them (in the 2nd phase), tasks take and dynamically upgrade version numbers, which optimizes unnecessary blocking. The conflicting operations are ordered according to versions, which is similar to *timestamp* algorithms [2, 32]. However, we associate versions with verlocks, not with transactions. Therefore all data accesses protected by verlocks are always made in the right order for the isolation property (the verlock requests with too high versions are simply delayed), unlike common 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.

More discussion of other related work on algorithms can be found in [34].

**Transaction models** Turning to the semantics of transactions, Chrysanthis and Ramamritham [6] have specified the broad spectrum of transactional models.

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. The above models are however presented abstractly, without being integrated with any language or calculus.

Vitek *et al.* [31] 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.

Berger and Honda [1] have used a variant of $\pi$-calculus to formalize the operational semantics of the standard two-phase commitment protocol for *distributed* transactions. This work however does not address local concurrency control (on a machine) and the isolation property.

## 7. CONCLUSION AND FUTURE WORK

The paper describes a language and runtime support for isolation-only, multithreaded transactions (tasks). The main idea of the paper is to avoid the need for rollback at runtime, which greatly simplifies the runtime system, allows tasks to perform arbitrary I/O operations, and also eliminates the risk of multiple restarts when many tasks compete for the same resource (since no task is aborted).

The runtime system requires however resources to be known *a priori*. Therefore, to make our language safe, we propose in this paper a type system that can verify resource declarations for the concurrency controller.

For clarity, we have 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 work on more robust approaches to implementing `isolated`.

The type system could be extended to 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, Peter Sewell, the Crystall project participants and the anonymous referees for helpful comments on drafts of this paper. This work was supported by Swiss NSF contract #21-67715.02 and Hasler Stiftung project DICS-1825.

## 8. REFERENCES

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

[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, 1996.

[4] A. P. Black, V. Cremet, R. Guerraoui, and M. Odersky. An equational theory for transactions. In *Proc. FSTTCS '03*, Dec. 2003.

[5] R. Bruni, H. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *Proc. POPL '05*, Jan. 2005.

[6] P. K. Chrysanthis and K. Ramamritham. ACTA: A framework for specifying and reasoning about transaction structure and behavior. In *ACM SIGMOD Conference on Management of Data*, 1990.

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

[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 *Proc. ESOP '99*, LNCS 1576, Mar. 1999.

[10] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proc. PLDI '03*, June 2003.

[11] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proc. POPL '96*, Jan. 1996.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, 2nd Ed.* Addison Wesley, 2000.

[13] M. H. Graham. Issues in real-time data management. Technical Report SEI-TR-17, 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 TOPLAS*, 16(6):1719–1736, Nov. 1994.

[15] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proc. OOPSLA '03*, 2003.

[16] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *Proc. PPoPP '05*, June 2005.

[17] 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.

[18] 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.

[19] S. Jagannathan and J. Vitek. Optimistic concurrency semantics for transactions in coordination languages. In *Proc. Coordination '04*, LNCS 2949, Feb. 2004.

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

[21] S. Mena, A. Schiper, and P. T. Wojciechowski. A step towards a new generation of group communication systems. In *Proc. Middleware '03*, LNCS 2672, 2003.

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

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

[24] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. POPL '96*, Jan. 1996.

[25] 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.

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

[27] *SAMOA*. `http://lsrwww.epfl.ch/samoa`.

[28] P. Sewell, P. T. Wojciechowski, and B. C. Pierce. Location-independent communication for mobile agents: a two-level architecture. In *Internet Programming Languages*, LNCS 1686, 1999.

[29] 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.

[30] Sun Microsystems. *EJB*. `http://java.sun.com/`.

[31] J. Vitek, S. Jagannathan, A. Welc, and A. L. Hosking. A semantic framework for designer transactions. In *Proc. ESOP '04*, LNCS 2986, March/April 2004.

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

[33] J. M. Wing, M. Faehndrich, J. G. Morrisett, and S. Nettles. Extensions to Standard ML to support transactions. In *Proc. ACM Workshop on ML and its Applications*, 1992.

[34] P. Wojciechowski, O. Rütti, and A. Schiper. SAMOA: A framework for a synchronisation-augmented microprotocol approach. In *Proc. IPDPS '04*, 2004.

[35] P. T. Wojciechowski. Isolation-only transactions by typing and versioning. Technical Report IC-2004-104, EPFL, School of Computer and Communication Sciences, Dec. 2004.