Coqa: Concurrent Objects with Quantized Atomicity

Yu David Liu Xiaoqi Lu Scott F. Smith

Department of Computer Science
The Johns Hopkins University
{yliu, xiaoqilu, scott}@cs.jhu.edu

Abstract

This paper introduces a new language model, *Coqa*, for deeply embedding concurrent programming into objects. Every program written in our language has built-in the desirable behaviors of atomicity, mutual exclusion, and race freedom. A key property of our model is the notion of quantized atomicity: *all* concurrent program executions can be viewed as being divided into quantum regions of atomic execution, greatly reducing the number of interleavings to consider. We justify our approach both from a theoretical basis by showing that a formal representation, KernelCoqa, has provable quantized atomicity properties, and by implementing CoqaJava, a Java extension incorporating all of the Coqa features.

1. Introduction

Programming languages have continually evolved to meet the programming demands of the times. Today it is arguable that the most significant new demand is to make languages that match the evolution to multi-core CPUs and the coming ubiquity of concurrent computation: nearly all programs will be concurrent. This paper is a reconsideration of the "right" concurrency model for tightly-coupled computations that can be easily deployable on multi-core CPUs. We begin this paper by enunciating our key concurrent language design principles.

1.1 Concurrent Language Design Principles

Design Principle 1: Good Concurrency Properties Preserved by Default Concurrency is particularly important to get right at this point in time as it is clear that future software development is going to rely more on multi-core CPUs: concurrency is fast becoming pervasive. Java was clearly *not* designed with pervasive concurrency in mind, it was designed for sequential programming, but with a standard bag of tricks to allow the critical small concurrent sections to be written. This was a reasonable choice for Java, since it was initialized to target portable Internet applications but not multi-core CPUs. Java requires programmers to *explicitly* declare the need of race freedom by using `synchronized`, with the implicit meaning that the *default* case is no use of these and thus no concern about concurrent access. In the coming era of multi-core CPUs, we believe the default mode here should be *inverted*: good properties such as race freedom, mutual exclusion, and atomicity should be *preserved* unless programmers *explicitly declare otherwise*.

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

Design Principle 2: Always* En Guarde *While Sharing For all concurrency models, one of the primary questions is whether a shared memory is assumed. Actor languages [1] take one extreme route of the non-shared-memory model: each actor has its own private store. This indeed makes sense for distributed programming but makes tightly-coupled computations very difficult to write.

On the other end of the spectrum is Java's "over-friendly" shared memory model. By default, object access by different Java threads is not protected, so that arbitrary interleaving between threads can happen. This makes concurrent Java programs very difficult to debug, which as a result introduces a large number of ugly complications including a very complex memory model [20].

We believe a good concurrent language model should take the middle route between the two extreme views above: we propose a shared memory model, but where different threads look after their own interests, possibly at the expense of other threads. In such a model, different threads do not take a "live and let live" attitude toward others and allow free sharing; instead they compete for (ephemeral) exclusive rights on the shared memory.

Design Principle 3: The Importance of Being Ubiquitously Atomic A distinct challenge of concurrent programming is that programs are difficult to reason about because there are an incredibly large number of interleaving possibilities across all possible runs of the program. Even though this is an inherent problem for any concurrent program, the situation today is much graver than it was, say, two decades

ago. With tightly coupled computation running on multi-core CPUs, data sharing between threads is much more common and the patterns are more complex. Ever-larger concurrent programs make the situation even more of a challenge. We strongly believe that some notion of *atomicity* – *i.e.* the property that a block of code can always be viewed as occurring atomically no matter what interleaving it is involved in – must be built-in to the next generation of concurrent object models.

Language support for atomicity is by no means news. This in fact is the primary goal of Software Transactional Memory (STM) [12, 30, 13, 26, 7, 3, 8]. These systems typically allow programmers to explicitly declare a block of code to be `atomic`. The underlying message of such a declaration however is that atomicity is still a luxury not a necessity – code blocks are not atomic unless explicitly declared to be. As a result, in a large number of STM systems [12, 30, 3, 8], code by default runs in non-transactional mode with zero atomicity guarantees, and the interleaving of this code with the transactional code in fact can break the atomicity of the latter, an unfortunate consequence known as weak atomicity [7]. It is our belief that code in next-generation languages should by default have some notion of atomicity inherent from the object model itself, not optional syntax at the mercy of programmers.

In this design principle, we are well aligned with Actors [1, 2]: there atomicity is preserved for each method because its execution once initiated does not depend on the state of other actors and each method is therefore trivially serializable.

Design Principle 4: Atomicity is Not Necessarily All or Nothing At first glance, one might think atomicity is either preserved or violated, and there is no middle ground that can be useful for programmers. Whether this is true depends on how the notion of atomicity is defined; different researchers have different definitions. But if we forget about these technical differences and think of atomicity more generally as reducing the chaos of too many interleaving possibilities, then some middle ground does indeed exist.

For instance, let us consider two pieces of code `X` and `Y`. If we know `X` in its entirety can always be viewed as occurring atomically no matter what interleaving it is involved in, and the same holds for `Y`, interleaving of the two can only have two possibilities, `XY` or `YX`, which is very useful for reasoning. However, suppose we are not that lucky, and `X` is known to contain two sub-regions each of which has the strong guarantee mentioned above but not the two combined together; have we lost all reasoning power? Obviously not, now the interleaving of `X` and `Y` will still only have three possibilities: X_aYX_b, YX_aX_b, and X_aX_bY. From the perspective of facilitating reasoning, we still have a significant improvement over the chaos of arbitrary interleavings.

It is with this notion as a guide that we have come up with our central principle of quantized atomicity, which we describe below.

Design Principle 5: Optimistic Atomicity is Not Always the Best Policy Programming language efforts to achieve atomicity largely fall into two categories: the *pessimistic blocking approach* (such as [9]) and the *optimistic rollback approach* (such as [12]). In the first approach, contentions are fundamentally eliminated as any contention-inducing operations will be blocked until the contention is resolved. In the optimistic rollback approach, execution of an atomic unit does not block, but at the end of the execution checks if any contention indeed happened. If so, the execution rolls back to the beginning of the atomic unit and retries. The atomic units in the optimistic approach are commonly called *transactions*.

The optimistic approach is the modus operandi of today's research on atomicity. One fundamental reason we think the optimistic approach does not serve as a good general programming model is it discriminates against certain forms of code, especially code with I/O. For instance, the need to roll back a network message sent across political boundaries opens up a Pandora's Box of problems. This shortcoming also directly clashes with our Design Principle 3 above: it is not possible to have a ubiquitous notion of atomicity in the presence of I/O if the optimistic approach is taken.

The optimistic and pessimistic approaches are compared in more detail in Sec. 2.3 below.

Design Principle 6: Put OO-Style Concurrency in OO Languages A related problem with Java not being designed from the start with concurrency front-and-center is how a non-object-based syntax and semantics is used for concurrent programming in Java. Language abstractions such as library class `Thread` and thread spawning via its `start` method, `synchronized` blocks and `atomic` blocks in various STM extensions of Java are not so different from what was used three decades ago in non-object-oriented languages [19]. A small and uniform core is favored as it makes reasoning easier, makes feature intervention is less likely, and the programmer's learning curve less steep.

1.2 Our Approach

In this paper, we describe a new object-oriented language, *Coqa* (for *C*oncurrent *o*bjects with *q*uantized *a*tomicity), which builds concurrency and atomicity-by-design deeply into the object model. We are more in the Actors camp than the STM camp in that we do not use rollback to enforce atomicity, but we present a programming model that is more direct than Actors which keeps the programmer from having to do "control flow hacking" just to write basic tasks. Our model also is not opposed to STM in the sense that it could be extended to a hybrid approach [29].

We model concurrent computation as a number of threads (in our terminology, *tasks*) competing to "capture" objects,

obtaining exclusive rights to them. A key property of Coqa programs is that object methods will often have complete atomicity over the whole method by design. But, for some tasks a method simply cannot reasonably be atomic, and our model allows programmers to relax this property by dividing a method into a small number of zones of atomicity, giving a model we call *quantized atomicity*. From a concurrent programming perspective the model has three desirable properties, from higher- to lower-level as follows:

Quantized atomicity Each method is composed of several discrete *quanta*, and execution of each quantum is serializable regardless of the interleaving of the actual execution.

Mutual exclusion within tasks Our language guarantees state change happens in a predictable way, even across different quanta of a task.

Race freedom No race conditions ever arise in object field access.

One of the main goals of our design is to significantly reduce the number of interleavings possible in concurrent program runs, to significantly ease the debugging burden for programs. If two pieces of code each have 100 steps of execution, reasoning tools would have to consider interleaving scenarios of 101^{100} possibilities; however, if the aforementioned 100 steps can be split into 3 atomic quanta, there are only 4^3 possibilities.

The contributions of this paper include:

- We define a new object model with inherently good properties of atomicity and mutual exclusion, rather than having bad default behavior and programmer declaration of isolated "good zones". Our model also is unambiguous about the guarantees it provides to programmers, the three properties above.
- We take advantage of basic object-oriented notions to express important concurrent programming primitives. In particular, different forms of message passing are all that is needed to express creating a separate task, local messaging within a task, and subtasking (a form of nested atomicity).
- We formalize the core language, and prove the model has the properties of quantized atomicity, mutual exclusion, and race freedom mentioned above.
- Lastly, we describe CoqaJava, a prototype translator implementing Coqa as a Java extension by simply replacing Java threads with our new forms of object messaging. The sequential core of the language remains unchanged.

2. Informal Overview

In this section, we informally introduce the key features of Coqa in an incremental fashion. We first describe a barebones model (Sec. 2.1) to illustrate the basic ideas, and then present the full model of Coqa in Sec. 2.2.

```
class BankMain {
  public static void main (String [] args) {
     Bank bank = new Bank();
     bank.openAccount("Alice", 1000);
     bank.openAccount("Bob", 2000);
     bank.openAccount("Cathy", 3000);
     bank -> transfer("Alice", "Bob", 300);   //(M1)
     bank -> transfer("Cathy", "Alice", 500); //(M2)
     bank -> openAccount("Dan", 4000);        //(M3)
  }
}

class Bank {
  public void transfer (String from, String to, int bal) {
     Status status = new Status();
     status.init();                                    //(A1)
     Account afrom = (Account)htable.get(from);        //(A2)
     afrom.withdraw(bal, status);                      //(A3)
     Account ato = (Account)htable.get(to);            //(A4)
     ato.deposit(bal, status);                         //(A5)
  }
  public void openAccount(String n, int bal) {
     htable.put(n, new Account(n, bal));
  }
  private HashTable htable = new HashTable();
}

class Account {
  public Account(String n, int b) {name = n; bal = b; }
  public void deposit(int b, Status s) {
    bal += b;
    s.append("Deposit " + b + " to Acc. " + name);
  }
  public void withdraw(int b, Status s) {
    bal -= b;
    s.append("Withdraw " + b + " from Acc. " + name);
  }
  private String name;
  private int bal;
}

class Status {
  public void init() {statusinfo = some time stamp info; }
  public void append(String s) {statusinfo.append(s);}
  public void print() {System.out.println(statusinfo); }
  private StringBuffer statusinfo = new StringBuffer();
}
```

Figure 1. A Banking Program: Version 1

The running example Throughout the section, we will use a simple example of basic banking operations, including account opening and balance transfer operations. Fig. 1 and Fig. 2 give the barebones version the source code we start with. Bank accounts are stored in a hash table, implemented in standard fashion with bucket lists. For instance, after the three accounts `Alice`, `Bob`, `Cathy` have been opened via the first four lines of the `main` method, a possible layout for objects in the hash table is pictured in Fig. 3. For brevity here

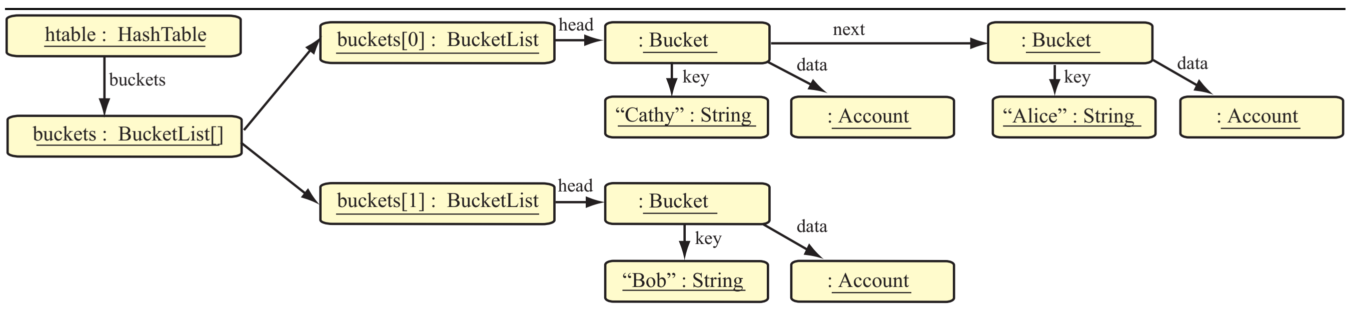

Figure 3. Hash Table: A UML Illustration

```
class HashTable {
  public Object get(Object key) {
     return buckets[hash(key)].get(key);
  }
  public void put(Object key, Object data) {
     buckets[hash(key)].put(key, data);
  }
  private int hash(Object key) {
     return key.hashcode() % 100;
  }
  private BucketList[] buckets = new BucketList[100];
}

class BucketList {
  public Object get(Object key) {
     for (Bucket b = head; b!= null; b = b.next())
         if (b.key().equals(key)) return b.data();
     return null;
  }
  public void put(Object key, Object data) {
     head = new Bucket(key, data, head);
  }
  private Bucket head;
}

class Bucket {
  Bucket(Object k, Object d, Bucket b)
     {key = k; data = d; next = b;}
  public Object key() {return key;}
  public Object data() {return data;}
  public Bucket next() {return next;}
  private Object key;
  private Object data;
  private Bucket next;
}
```

Figure 2. HashTable

we present simplified code which omits checks for duplicate keys, unlocatable accounts, and overdrafts.

2.1 The Barebones Model

The concurrency unit and its lifecycle The concurrency unit in our language is a *task*. This refers to a unit of execution that can potentially be interleaved with other units. Tasks are closely related to (logical) threads, but tasks come with inherent atomicity properties not found in threads and we coin a new term to reflect this distinction.

Since we are building an object-based language, the syntax for task creation is also object based: tasks are created by simply sending asynchronous messages to objects. The essence of asynchronous messaging is that the sender does not expect any result from the receiver, and thus it is a good point for parallelizing the program. For example, in Fig. 1, the top-level `main` method starts up two concurrent balance transfers by the invocations in lines `M1` and `M2`. Syntax `bank->transfer("Alice", "Bob", 300)` indicates an asynchronous message `transfer` sent to object `bank` with indicated arguments. Asynchronous message send returns immediately, so the sender can continue, and a new task is created to execute the invoked method. This new task terminates when its method is finished.

Tasks are selfish According to Design Principle 2 in Sec. 1, tasks selfishly *compete* for objects in the heap. The basic idea is simple. All Initially instantiated objects are not held by any task. We may say they are *free* in the heap. Whenever a task accesses a free object, it selfishly *captures* it. At any moment of execution, a task can be viewed as having selfishly captured a set of objects (we say these objects are in the task's *capture set*). Objects in the capture set are all freed when the task ends. If a task intends to access an object already captured by some other task, it is blocked at that point of execution until the needed object is freed.

Capturing is a blocking mechanism, but unlike Java where programmers need to explicitly specify what to lock and what not to lock, the capture and blocking of objects is fundamentally built into Coqa.

The timing and scope of capture The description of capture above is not complete: we still must specify *when* an object is captured and *how selfish* captures are.

In regard to timing, consider `transfer` method of Fig. 1. When the programmer writes line `(A2)`, his/her intention of selfishly capturing the `HashTable` object referenced by `htable` is expressed: synchronous messaging at a high level expresses the fact that the task will accessing the receiver. However, this view is more conservative than is needed: what really matters is exclusive access to (mutable) data, *i.e.* the underlying fields. So, in Coqa we define the capture points to be the field access points.

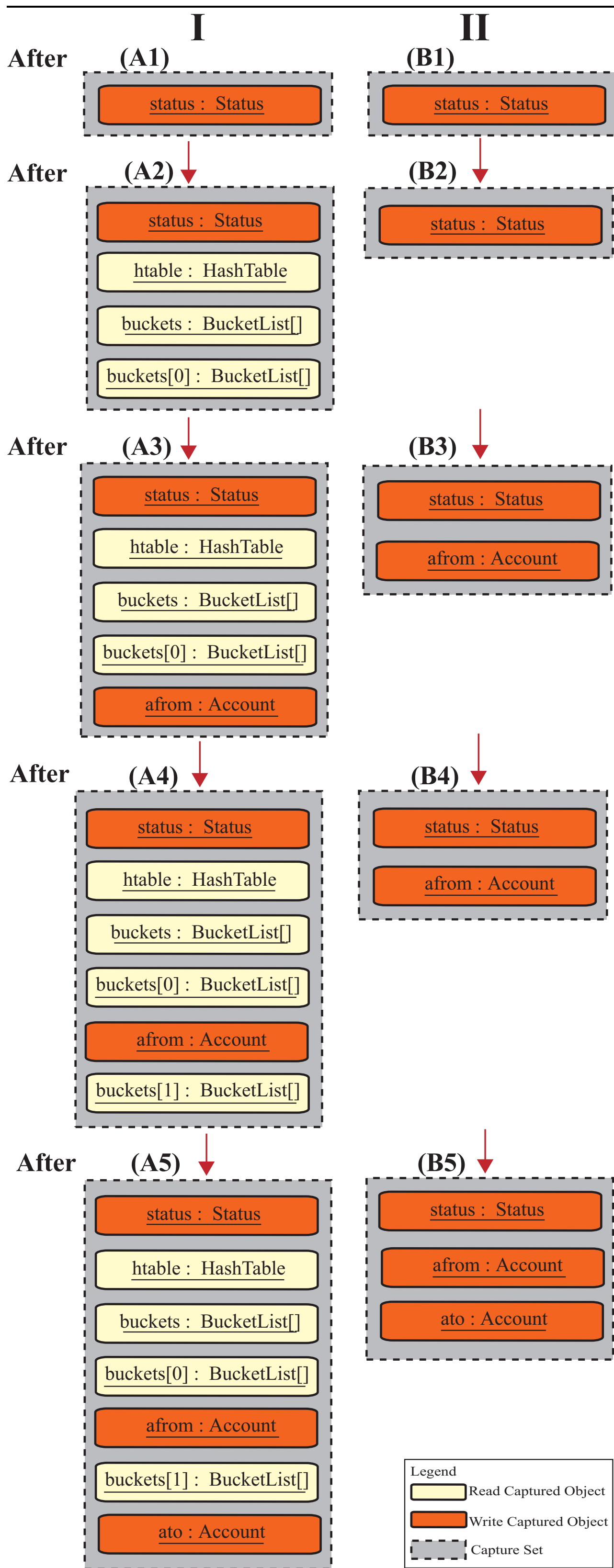

Figure 4. Execution Sequence for the Task Created by Fig. 1 Line `M1`

The second question is related to field access itself. The selfish model we have thus far described treats read and write access uniformly. If this were so, any `transfer` task must exclusively hold the `HashTable` object `htable` in line `(A2)` until the entire `transfer` task is completed. As a further optimization our notion of capture is further refined to use the standard two-phase non-exclusive read lock and exclusive write lock [10] approach. When an object's field is read, the object is said to be *read captured*; when the field is written, the object is said to be *write captured*. The same object can be read captured by multiple tasks at the same time, but to be write captured, the object has to be exclusively owned, *i.e.* not read captured or write captured by another other task.

One possible execution sequence of the task created by line `(M1)` of Fig. 1 is illustrated in the first column of Fig. 4. As the execution proceeds, more and more accessed objects are captured via field reads (yellow boxes) or writes (red boxes). For space reasons, we have omitted `String` objects which are immutable and thus always read captured.

The two-phase locking strategy increases the opportunity for parallelism, but it is known to have one disadvantage: one form of deadlock is possible as a result. Consider when tasks `(M1)` and `M2` are both withdrawing money from `Alice`'s account. Both have read from her `Account` object the current balance, but neither has written the new balance yet . The `Account` object of `Alice` would have read locks from both `(M1)` and `(M2)`, and neither party can add a write lock to it, and so neither party can proceed. To solve this matter, our language allows programmers to declare a class with tightly coupled read/write as an `exclusive` class. Exclusive classes have only one lock for either read or write, and parallel read is prevented. For instance, the `Account` class could have been declared as **exclusive class** `Account {...}` and that would have prevented deadlock from arising.

Properties The barebones model has several good properties for tasks running in parallel: atomicity, mutual exclusion, and race freedom.

Let us consider two tasks that run in parallel, say `X` and `Y`. Intuitively, if `X` and `Y` do not read/write any common object at all, they can be arbitrarily interleaved without affecting each other. If they do need to access the same objects, note that our philosophy of capture is that they need to compete for them. At any particular time and no matter how `X` and `Y` have interleaved up to this moment, we know the objects `X` have accessed must either be read captured or write captured by `X`. It is obvious `Y` must not have write captured any of these objects since `X` becomes alive. In other words, all of these objects are not mutated by `Y`. This fact has several appealing consequences for the interleaving tasks of `X` and `Y`:

Atomicity : Since `Y` has no (mutation) effect on the objects `X` cares about, all computations performed by `Y` can be equivalently viewed as non-existent from `X`'s perspective. Task `X` can then be viewed as running alone. The same

can be said about Y. Together this implies that the execution of X and Y can be serialized.

Mutual Exclusion : Following the same reasoning, X and Y have no (mutation) effect on each other's captured objects.

Race Freedom : As a special case of mutual exclusion, X and Y does not race to access any object field, except that both may have been harmlessly reading from the same object field.

In our language, the only way to create a new task is to send asynchronous messages to an object, and since our language allows any object to receive asynchronous messages provided there is such a method, it follows that running any two methods in parallel preserves the aforementioned properties. This is an important observation since it correlates the properties of a highly dynamic construct (tasks) to what programmers have in mind while coding, the static construct (methods). For instance, it helps programmers to be aware that a method is in fact a complete unit of mutual exclusion, a natural connection that helps programmers to express their intentions. Our notion of object mutual exclusion is much stronger than when Java programmers declare a method to be synchronized: this declaration only guarantees the object with the method is itself not mutated by other threads, while we are guaranteeing the property for *all* objects directly or indirectly accessed at run time by the method, many of which may be unknown to the caller.

2.2 Subtasking: Open Nesting with Blocking

The barebones model admits significant parallelism if most of the object accesses are read accesses, as read capture does not prevent concurrent access. Long waits are possible however when frequent writes are needed. For instance, consider the parallel execution of the two tasks spawned by (M1) and (M3). Let us suppose when the account of Dan is added, it will become the head Bucket of bucket[0] in Fig. 3. The task started up by (M1) will label the object of bucket[0] (of type BucketList) to be read captured. This will completely block the task created by (M3), since executing it will demand exclusive write capture. M3, the task of adding Dan as a new account – can only be achieved after the completion of M1, the totally unrelated task of transferring money from Alice to Bob. Intuitively, there at least should be some parallelism in running the two tasks.

The source of this problem is the transfer task created by (M1) always remembers it has accessed the hash table (and hence bucket[0]) *throughout the lifecycle of the task*. This only makes sense if the programmer indeed needs to make sure the hash table is never mutated by other tasks throughout the duration, to guarantee complete atomicity of its behavior. In the example here, this is hardly necessary; there is nothing wrong with the fact that when the (M1) locates the account of Alice, the account of Dan does not exist, but when later (M1) locates the account of Bob, the account of Dan has already been opened.

The idea of subtasking To get around the previous shortcoming, our language allows programmers to spawn off the access of HashTable object (and all objects it indirectly accesses) as a new *subtask*. The high-level meaning behind a subtask is that it achieves a relatively independent goal; its completion signals a partial victory so that the resources (in this case captured objects) used to achieve this subtask can be freed.

In terms of syntax, we can change the source code of transfer in Fig. 4 to the following Fig. 5 which makes the two HashTable accesses run as subtasks. The only change is the dot (.) notation for synchronous messaging is changed to ⇒ for subtask creation messaging. To distinguish the two forms of synchronous messaging, the original dot-notation is hereafter called *local* synchronous messaging since its execution stays within the current task and does not start a new subtask.

```
public void transfer (String from, String to, int bal) {
   Status status = new Status();
   status.init();                                    //(B1)
   Account afrom = (Account)htable⇒get(from);  //(B2)
   afrom.withdraw(bal, status);                      //(B3)
   Account ato = (Account)htable⇒get(to);      //(B4)
   afrom.deposit(bal, status);                       //(B5)
}
```

Figure 5. Bank's transfer method: Version 2

This is still a synchronous invocation, *i.e.*, the task executing transfer waits until its subtask executing get returns a result. But the key difference is the subtask keeps a *separate* capture set, and that capture set is freed when the subtask is finished. This can be illustrated in the second column of Fig. 4. When the method invocation htable⇒get(from) completes, the task of opening an account for Dan can now access the HashTable, rather than requiring to wait until M1 has completely finished.

A subtask is also a task in the sense that it prevents arbitrary interleaving. The change at line (B2) admits some interleaving between task (M1) and (M3) that was not allowed before, but it does not mean that arbitrary interleaving can happen; for example, if M1 is in the middle of a key lookup M3 still *cannot* add a new bucket. We will discuss such concurrency properties in the presence of subtasking later in this section.

Subtasking is a dual to open nesting in transaction-based systems [24, 7]. Open nesting is used to nest a transaction inside another transaction, where the nested transaction can commit before the enclosing transaction runs to completion. While the mechanism of open nesting of transactions can be

summarized as early commit, subtasking can be summarized as early release.

Capture set inheritance One contentious issue for open nesting is the case where a nested transaction and the transactions enclosing it both need the same object. For instance in Atomos [7], the issue is circumvented by restricting the read/write sets to be disjoint between the main and nested transaction. When the same issue manifests itself in the scenario of subtasking, the question is, "how can a subtask access objects already captured by its enclosing task(s)?"

We could in theory follow Atomos and add the restriction that a subtask never accesses an object held by its enclosing tasks. This however would significantly reduce programmability. Let us consider the example of the `Status` object in the `transfer` method. From the programmer's view, this object keeps track of the status of the entire `transfer` method. When the `Account` objects of `Alice` and `Bob` are accessed, some status information needs to be appended to the `Status` object, which in our case is already captured by the `transfer` task. Had we disallowed the access of `Status` from the `HashTable` task, the program would deadlock when `s.append()` was invoked inside the `Account` objects.

This case in Coqa is perfectly legal. In fact, we believe the essence of having a subtasking relationship between a parent and a child is that the parent should generously share its resources (its captured objects) with the child. Observe that the relationship between a task and its subtask is a synchronous one: there is no concern of interleaving between a task and its subtask. The subtask should thus be able to access all the objects held by its direct or indirect "parent" tasks without introducing any unpredictable behaviors. For the program in Fig. 5, after line `(B3)`, the `Status` object will be mutated to reflect a withdraw for `Alice`.

Quantized atomicity The presence of subtasking in a method weakens its atomicity properties: the objects captured by the subtask in fact can serve as a communication point between different tasks. For a more concrete example, consider two tasks `X` and `Y` running in parallel. Suppose task `X` creates a subtask, say `Z`, and in the middle of task `Z`, some object `o` is read captured. According to the definition of subtasking, `o` will be freed at the end of `Z`. `Y` can subsequently write capture it. After `Y` ends, suppose `X` read captures `o`. Observe that object `o`'s state has been changed since its subtask `Z` had previously read `o`. Task `X` thus cannot assume it is running in isolation, and so it is not fully atomic.

One the other hand, some tasks simply *should not* be considered wholly atomic because they are *fundamentally* needing to share data with other tasks, and for this case it is simply *impossible* to have full atomicity over the whole task. In fact, the main reason why a programmer wants to declare a subtask is to open a communication channel with other tasks. This fact was directly illustrated in the subtasking example at the start of this subsection. Fortunately even in such a situation, some very strong properties still hold, which we will rigorously discuss in Sec. 3.3:

Quantized Atomicity: For any task, its execution sequence consists of a sequence of atomic regions, the *atomic quanta*, demarcated by the task and subtask creation points.

Selective Mutual Exclusion: Still holds as in Sec. 2.1, with the one exception of sharing allowed with subtasks.

Race Freedom: Still holds as in Sec. 2.1.

The quantized atomicity property is weaker than the atomicity property of the barebones system without subtasks, but as long as quantized atomicity is used only when it is really needed, the atomic quanta will each be large and significant atomicity can be achieved. In reality, what matters is not that the entire method must be atomic, but that the method admits drastically limited interleaving scenarios. Quantized atomicity aims to strikes a balance between what is realistic and what is reasonable.

2.3 The Design Choice of Blocking vs Rollback

Coqa uses pessimistic blocking via locks to preserve atomicity. In this section we contrast our approach with the STM approach of transactions with optimistic execution and rollback/commit.

The rollback approach has known difficulties on rolling back certain forms of computations, I/O in particular. GUI applications are classic examples where concurrency is very helpful, but where I/O is pervasive and thus rollback will be infeasible since output already displayed cannot easily be taken back. In the approach we are taking, these examples are particularly bad: we are proposing a language where atomicity is pervasive through *all* computations, and it would be very difficult to apply a transactional technique to such a universal setting due to problems such as I/O. So, the fundamental fact is while transactions have great appeal, they just do not work for our full purposes here (in fact, it would be useful to use STM in special cases in our model as an optimization *e.g.* in regions where there is no I/O, just not everywhere).

Rollback also may not be as easy as simply discarding the read/write set and retrying. In realistic STM languages, it is common to support primitives such as what needs to be compensated at abort time or at violation time (`AbortHandler`, *etc.* in [7] and `onAbort` *etc.* methods in [24]) as user-definable handlers. These handlers themselves may introduce races or deadlocks, so some of the appeal of the approach is lost; see [24]. Related to this point is how the implementation of transaction-based systems are quite complex. Earlier efforts typically needed some hardware support, such as [12, 7]. Pure software source-to-source translations do exist, such as [26, 24]

In terms of performance there have been no clear studies that we know of comparing the approaches. Generally there

messaging	why you should use it	why you should not use it too much
`o .m(v)`	strongly promotes mutual exclusion and atomicity	no speed up; potential for deadlock
`o ⇒ m(v)`	promote parallelism by encouraging early release	loss of whole-method atomicity
`o -> m(v)`	promote parallelism by starting up a new task	no return value; no capture set inheritance

Figure 6. The three messaging mechanisms and their relative strengths

is a close correspondence between when transaction-based systems will need to rollback and when Coqa will need to block: rollback and blocking happens *only* in cases where there is contention. For this reason there is some cancellation of overhead: the overhead of blocking aligns with the overhead of rollback. If there is no contention, the comparison is between maintenance of locks on objects in the blocking approach, and duplicating of shared state , and performing validation at the end of the transaction to make sure different read/write sets do not lead to contention.

Blocking systems like ours are more likely to deadlock due to the increased amount of mutual exclusion required. This is the biggest disadvantage of the blocking approach, but the programming burden has shifted from dealing with all of the other problems to instead focus on deadlock detection and the code refactoring needed to fix it. It significantly raises the importance of developing quality deadlock detection tools for both compile-time and run-time analysis to better discover and correct deadlocking code. This also shows another fundamental difference between the blocking and rollback approaches: blocking requires a more structured approach to programming, not every program will work. The history of programming languages has shown that using a more structured programming methodology is a feature, not a bug.

The counterpart to deadlock in transaction-based systems is *livelock*, the case where rollbacks resulting from contention might result in further contentions and further rollbacks, *etc*. How frequently livelocks occur is typically gauged by experimental methods. There is nothing wrong with this, but it disqualifies transaction-based systems from critical systems where impasse is fundamentally not acceptable.

One advantage of rollback-based systems is they often have a totality to atomicity: The sequence of events can be viewed as all happening or none happening, never half-happening [17]. While this is a desirable property, it can never hold in Coqa because of the combination of the infeasibility of rolling back I/O and the requirement to factor all execution into atomic quanta. A good overview of the pros and cons of blocking and rollback appears in [29].

2.4 Be a Happy Coqa Programmer

Here we summarize why Coqa is a good programming platform.

A simple model with choices Coqa has a very simple object syntax: the only difference from the Java object model is a richer syntax to support object messaging, and this new syntax also encompasses thread spawning. So the net difference in the syntax is near-zero. We summarize the three different forms of object messaging, along with the cases where they are the most appropriate, in Fig. 6. If we imagine the `HashTable` object as a service, one feature of Coqa is how the *client* gets to decide what atomicity is needed. For instance, both the `transfer` method in Fig. 1 and in Fig. 5 are valid programs, depending on what the programmer believes is needed for the `transfer` method, full atomicity or quantized atomicity. As another example, if the programmer does not need to account for random audits which would sum the balance of all accounts, the programmer could decide to have `withdraw` and `deposit` run as subtasks as well, resulting the following program:

```
public void transfer (String from, String to, int bal) {
   Status status = new Status();
   Account afrom = (Account)htable⇒get(from);
   afrom⇒withdraw(bal, status);
   Account ato = (Account)htable⇒get(to);
   ato⇒deposit(bal, status);
}
```

Figure 7. `Bank`'s `transfer` Method: Version 3

This version of `transfer` will result in more concurrency between `(M1)` and `(M2)`, since right after `(M1)` withdraws from `Alice`'s account, the bank will become available for `M2` to deposit into.

Avoiding deadlocks There are two forms of deadlock arising in Coqa. The first form was described in Sec. 2.1: deadlock inherent in two-phase locking. The second form is the standard cyclically dependent deadlock. Consider the `transfer` example in Fig. 1. If there are two tasks where one needs to transfer a balance from `Alice` to `Bob`, while the other from `Bob` to `Alice`, deadlock happens if the first task write captures `Alice`'s `Account` object and waits for `Bob`'s `Account` object, while the second task write captures `Bob`'s `Account` object and wait for `Alice`'s `Account` object.

The two forms of deadlock can be removed by refactoring programs. For instance, the first form of deadlock can be avoided by declaring a class to be `exclusive`, while the second form of deadlock does not exist in Fig. 7, a pro-

gram with *de facto* equivalent functionality. Deadlock-free programming is thus converted to the question of how a programmer can become aware of the existence of (potential) deadlocks.

There are many static and dynamic analysis techniques and tools to ensure deadlock freedom; for an overview, see [27]. A sound technique needs to make sure no deadlock is possible for *all* possible interleaving scenarios. The precision of static techniques are reduced due to the combinational explosion of interleaving. Note that since all Coqa code observes quantized atomicity, stronger analysis results are likely to be achievable over Coqa programs because interleaving has been reduced.

I/O atomicity Coqa supports two forms of I/O, either by invoking the I/O methods as local sends, or via subtasking. In the former, `System.out` is an `exclusive` object that needs to be write captured. That way, when one task needs to print `System.out.println("hello")`, the object `System.out` will be exclusively held by the task, and the output will not be interrupted by other tasks. In the case where the I/O devices support concurrent access by different tasks, such as a low-level socket on a window system, the device can be messaged as a subtask which will allow sharing of the I/O channel between tasks.

3. Formalization

In this section, we present a formal treatment of Coqa via a small kernel language called KernelCoqa. We first present the syntax and operational semantics of KernelCoqa, then a proof of quantized atomicity for KernelCoqa using the operational semantics, as well as other interesting corollaries.

3.1 KernelCoqa Syntax

P	::=	$\overrightarrow{cn \mapsto \langle l; Fd; Md \rangle}$	*program/classes*
Fd	::=	$\overline{fn}$	*fields*
Md	::=	$\overrightarrow{mn \mapsto \lambda x.e}$	*methods*
e	::=	$\mathbf{null} \mid x \mid cst \mid \mathbf{this}$	
		$\mid \mathbf{new}\ cn$	*instantiation*
		$\mid fn \mid fn = e$	*field access*
		$\mid e.mn(e)$	*local invocation*
		$\mid e \rightarrow mn(e)$	*task creation*
		$\mid e \Rightarrow mn(e)$	*subtask creation*
		$\mid \mathbf{let}\ x = e\ \mathbf{in}\ e$	*continuation*
l	::=	$\texttt{exclusive} \mid \epsilon$	*capture mode*
cst			*constant*
cn			*class name*
mn			*method name*
fn			*field name*
x			*variable name*

Figure 8. Language Abstract Syntax

We first define some basic notation used in our formalization. We write $\overline{x_n}$ as shorthand for a set $\{x_1, \ldots, x_n\}$, with empty set denoted as $\emptyset$. $\overrightarrow{x_n \mapsto y_n}$ is used to denote a mapping $\{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$, where $\{x_1, \ldots x_n\}$ is the domain of the mapping, denoted as $\mathsf{dom}(H)$. We also write $H(x_1) = y_1, \ldots, H(x_n) = y_n$. When no confusion arises, we drop the subscript n for sets and mapping sequences and simply use $\overline{x}$ and $\overline{x \mapsto y}$. We write $H\{x \mapsto y\}$ as a mapping update: if $x \in \mathsf{dom}(H)$, H and $H\{x \mapsto y\}$ are identical except that $H\{x \mapsto y\}$ maps x to y; if $x \notin \mathsf{dom}(H)$, $H\{x \mapsto y\} = H, x \mapsto y$. $H\backslash x$ removes the mapping $x \mapsto H(x)$ from H if $x \in \mathsf{dom}(H)$, otherwise the operation has has no effect.

The abstract syntax of our system is shown in Fig. 8. KernelCoqa is an idealized object-based language with objects, messaging, and fields. A program P is composed of a set of classes. Each class has a unique name *cn* and its definition consists of sequences of field (*Fd*) and method (*Md*) declarations. To make the formalization feasible, many fancier features are left out, including types and constructors.

Besides local method invocations via the usual dot (`.`), synchronous and asynchronous messages can be sent to objects using $\Rightarrow$ and $\rightarrow$, respectively.

3.2 Operational Semantics

Our operational semantics is defined as a contextual rewriting system over states $S \Rightarrow S$, where each state is a triple $S = H, N, T$ for H is the object heap, N a task ancestry mapping, and T the set of parallel tasks. Every task in turn has its local evaluation context $\mathbf{E}$. The relevant definitions are given in Fig. 9. A heap H is the usual mapping from objects o to field records tagged with their class name *cn*. In addition, an object on the KernelCoqa heap has two capture sets, R and W, for recording tasks that have read-captured or write-captured this object, respectively. A field store F is a standard mapping from field names to values. A task is a triple consisting of the task ID t, the object γ this task currently operates on and an expression e to be evaluated. N is a data structure which maps subtasks to their (sole) parent tasks. This is needed to allow children to share objects captured by their parent. We also extend expression e to include value v and two runtime expressions $\mathbf{wait}\ t$ and $e \uparrow e$.

The complete single-step evaluation rules are presented in Fig. 10. In this presentation, we use $e; e'$ as shorthand for $\mathbf{let}\ x = e\ \mathbf{in}\ e'$ where x is a fresh variable. These rules are implicitly defined over some fixed program P. Some of the rules in Fig. 10 have labels after the rule names which are used in subsequent Definitions and Lemmas below.

The INVOKE rule is used for local synchronous messaging, signified by dot (`.`) notation. Evaluation of a local synchronous message is interpreted a standard function application of the argument v to the method body of mn.

Rule TASK creates a new independent task via asynchronous messaging. The creating task continues its computation, and the newly created task runs concurrently with its

$$\textsc{TContext1}\quad \frac{H, N, T_1 \Rightarrow H', N', T_1'}{H, N, T_1 \parallel T_2 \Rightarrow H', N', T_1' \parallel T_2}$$

$$\textsc{TContext2}\quad \frac{H, N, T_2 \Rightarrow H', N', T_2'}{H, N, T_1 \parallel T_2 \Rightarrow H', N', T_1 \parallel T_2'}$$

$$\textsc{Set}\quad \frac{H(\gamma) = \langle cn; R; W; F\rangle \qquad H' = H\{\gamma \mapsto \langle cn; R; W \cup \{t\}; F\{fn \mapsto v\}\rangle\} \text{ if } R \subseteq \mathbf{ancestors}(N,t) \text{ or } W \subseteq \mathbf{ancestors}(N,t)}{H, N, \langle t; \gamma; \mathbf{E}[fn = v\,]\rangle \Rightarrow H', N, \langle t; \gamma; \mathbf{E}[\,v\,]\rangle}$$

$$\textsc{Get}\quad \frac{F(fn) = v \qquad H' = \begin{cases} H\{\gamma \mapsto \langle cn; R; W \cup \{t\}; F\rangle\}, & \text{if } l = \texttt{exclusive} \text{ and } R \subseteq \mathbf{ancestors}(N,t) \text{ or } W \subseteq \mathbf{ancestors}(N,t) \\ H\{\gamma \mapsto \langle cn; R \cup \{t\}; W; F\rangle\}, & \text{if } l = \epsilon \text{ and } W \subseteq \mathbf{ancestors}(N,t) \end{cases} \quad H(\gamma) = \langle cn; R; W; F\rangle \qquad P(cn) = \langle l; Md; Fd\rangle}{H, N, \langle t; \gamma; \mathbf{E}[\,fn\,]\rangle \Rightarrow H, N, \langle t; \gamma; \mathbf{E}[\,v\,]\rangle}$$

$$\textsc{This}\quad H, N, \langle t; \gamma; \mathbf{E}[\,\mathbf{this}\,]\rangle \Rightarrow H', N, \langle t; \gamma; \mathbf{E}[\,\gamma\,]\rangle$$

$$\textsc{Let}\quad H, N, \langle t; \gamma; \mathbf{E}[\,\mathbf{let}\ x = v\ \mathbf{in}\ e\,]\rangle \Rightarrow H, N, \langle t; \gamma; \mathbf{E}[\,e\{v/x\}\,]\rangle$$

$$\textsc{Return}\quad H, N, \langle t; \gamma; \mathbf{E}[\,v \uparrow o\,]\rangle \Rightarrow H, N, \langle t; o; \mathbf{E}[\,v\,]\rangle$$

$$\textsc{Inst}\quad \frac{P(cn) = \langle l; Fd; Md\rangle \qquad H' = H\{o \mapsto \langle cn; \emptyset; \emptyset; \biguplus_{fn \in Fd} \{fn \mapsto \mathbf{null}\}\rangle\},\ o \text{ fresh},}{H, N, \langle t; \gamma; \mathbf{E}[\,\mathbf{new}\ cn\,]\rangle \Rightarrow H, N, \langle t; o; \mathbf{E}[\,o\,]\rangle}$$

$$\textsc{Invoke}\quad \frac{H(o) = \langle cn; R; W; F\rangle \qquad P(cn) = \langle l; Fd; Md\rangle \qquad Md(mn) = \lambda x.e}{H, N, \langle t; \gamma; \mathbf{E}[\,o.mn(v)\,]\rangle \Rightarrow H, N, \langle t; o; \mathbf{E}[\,e\{v/x\} \uparrow \gamma\,]\rangle}$$

$$\textsc{Task}(t, \gamma, mn, v, o, t')\quad \frac{t' \text{ fresh}}{H, N, \langle t; \gamma; \mathbf{E}[\,o \rightarrow mn(v)\,]\rangle \Rightarrow H, N, \langle t; \gamma; \mathbf{E}[\,\mathbf{null}\,]\rangle \parallel \langle t'; o; \mathbf{this}.mn(v)\rangle}$$

$$\textsc{SubTask}(t, \gamma, mn, v, \gamma, t')\quad \frac{N' = N\{t' \mapsto t\} \qquad t' \text{ fresh}}{H, N, \langle t; \gamma; \mathbf{E}[\,o \Rightarrow mn(v)\,]\rangle \Rightarrow H, N, \langle t; \gamma; \mathbf{E}[\,\mathbf{wait}\ t'\,]\rangle \parallel \langle t'; o; \mathbf{this}.mn(v)\rangle}$$

$$\textsc{TEnd}(t)\quad \frac{H' = \biguplus_{H(o) = \langle cn; R; W; F\rangle} (o \mapsto \langle cn; R \backslash t; W \backslash t; F\rangle) \qquad N(t) = \mathbf{null}}{H, N, \langle t; \gamma; v\rangle \Rightarrow H', N, \epsilon}$$

$$\textsc{STEnd}(t, v, t')\quad \frac{H' = \biguplus_{H(o) = \langle cn; R; W; F\rangle} (o \mapsto \langle cn; R \backslash t; W \backslash t; F\rangle) \qquad N(t) = t'}{H, N, \langle t; \gamma; v\rangle \parallel \langle t'; \gamma'; \mathbf{E}[\,\mathbf{wait}\ t\,]\rangle \Rightarrow H', N \backslash t, \langle t'; \gamma'; \mathbf{E}[\,v\,]\rangle}$$

Figure 10. KernelCoqa Operational Semantics Rules

creating task. It may look like that the message mn is just resent to the target object o in TASK. But actually, a new task t' is created and the asynchronous message send becomes a local synchronous message in this newly-created task t'.

The SUBTASK rule creates a subtask of the current task via synchronous messaging. The creating task (the parent) goes in $kwwait$ state until the newly created subtask (the child) completes and returns. Therefore, a task can have at most one subtask active at any given time. The child-parent relationship is recorded in N.

When a task or a subtask finishes, all objects they have captured during their executions are freed. A subtask also needs to release its parent so it may resume execution and the mapping from the subtask to its parent is removed from N. The TEND and STEND are rules for ending a task and a subtask, respectively.

Before discussing the rules for object capture, let us first introduce the definition of the *ancestors* of a (sub)task, the set consisting of the (sub)task itself, its parent, its parent's parent, etc:

$$\mathbf{ancestors}(N,t) = \begin{cases} \{t\}, \text{if } N(t) = \mathbf{null} \\ \{t\} \cup \mathbf{ancestors}(N,t'), \text{if } N(t) = t' \end{cases}$$

The two capture sets, R and W of an object are checked and updated lazily: when a task actually accesses a field of an object. The check/capture policy is implemented in rules SET and GET.

H	::=	$\overrightarrow{o \mapsto \langle cn; R; W; F\rangle}$	*heap*
F	::=	$\overrightarrow{fn \mapsto v}$	*field store*
T	::=	$\langle t; \gamma; e\rangle \mid T \parallel T'$	*task*
N	::=	$\overrightarrow{t \mapsto t'}$	*subtasking relationship*
R, W	::=	$\overline{t}$	*read/write capture set*
γ	::=	$o \mid \mathbf{null}$	*current executing object*
v	::=	$cst \mid o \mid \mathbf{null}$	*values*
e	::=	$v \mid \mathbf{wait}\ t$	*extended expression*
		$\mid e \uparrow e \mid \ldots$	
$\mathbf{E}$	::=	$\bullet \mid fn = \mathbf{E}$	*object evaluation context*
		$\mid \mathbf{E}.m(e) \mid v.m(\mathbf{E})$	
		$\mid \mathbf{E} \rightarrow m(e) \mid v \rightarrow m(\mathbf{E})$	
		$\mid \mathbf{E} \Rightarrow m(e) \mid v \Rightarrow m(\mathbf{E})$	
		$\mid \mathbf{let}\ x = \mathbf{E}\ \mathbf{in}\ e$	
o			*object ID*
t			*task ID*

Figure 9. Dynamic Data Structures

The SET rule specifies that a task t can write capture object γ, the current object t is operating on, if all write capturers or all read capturers of γ are t's ancestors. GET checks the class exclusion label of γ first. If γ requires exclusive capture to access it, task t has to write capture γ before the read, which is similar to how SET works. If not, only read capture is needed. Task t can read capture γ if all of γ's write capturers are t's ancestors. It is worthwhile to emphasize that a subtask can both capture objects independently, and inherit objects already captured by its ancestors.

When a task cannot capture an object it needs, it is implicitly *object-blocked* on the object until it is entitled to capture it—the SET/GET rule cannot progress. The formal definition of object-blocked will be given in Sec. 3.3. Note that in this presentation we will not address the fairness of capture (or other fairness properties), but in a full presentation and in an implementation the unblocking should be fair so as to never starve a blocked task where the object was unblocked infinitely often.

Other rules in Fig. 10 are standard. For instance, object instantiation rule INST creates a new object, initializes all fields to be **null** and its R and W are initialized to $\emptyset$.

3.3 Atomicity Theorems

Here we formally establish the informal claims about KernelCoqa: quantized atomicity, mutual exclusion in tasks, and race freedom. Rigorous proofs are provided in the long version of this paper [18]. The key Lemma is the Bubble-Down Lemma, Lemma 2, which shows that adjacent steps of a certain form in a computation path can be swapped to give an "equivalent" path, a path with the same I/O behavior. Then, by a series of Bubblings, each quantum of steps can be bubbled to all be adjacent in an equivalent path, showing that the quanta are serializable: Theorem 1. The technical notion of a quantum is the *pmsp* below, a *pointed maximal sub-path*. These are a series of local steps of one task with a nonlocal step at the end, which may be embedded (spread through) in a larger concurrent computation path. We prove in Theorem 1 that any path can be viewed as a collection of *pmsp*'s, and all *pmsp*'s in the path are serializable and thus the whole path is.

Definition 1 (Object State). *Recall the global state is a triple $S = H, N, T$. The object state for o, written s_o, is defined as $H(o)$, the value of the object o in the current heap H, or* **null** *if $o \notin dom(H)$.*

We write step $st = (S, r, S')$ to denote a transition $S \Rightarrow S'$ by rule r of Figure 10. We let $change(st) = (s_o, r, s'_o)$ denote the fact that the begin and end heaps of step st differ at most on their state of o, taking it from s_o to s'_o. Similarly, the change in two adjacent steps which changes at most two objects is represented as $change(st_1 st_2) = ((s_{o_1}, s_{o_2}), r_1 r_2, (s'_{o_1}, s'_{o_2}))$.

Definition 2 (Local and Nonlocal Step). *A step $st = (S, r, S')$ is a* local *step if r is one of the local rules: either* GET, SET, THIS, LET, RETURN, INST *or* INVOKE. *st is a* nonlocal *step if r is one of nonlocal rules: either* TASK, SUBTASK, TEND *or* STEND.

Every nonlocal rule has a label given in Fig 10. For example, the TASK rule has label $\text{TASK}(t, \gamma, mn, v, o, t')$ meaning asynchronous message mn was sent from object γ in task t to another object o in a new task t', and the argument passed was v. These labels are used as the observable; the local rules have no labels.

Lemma 1. *In any given local step st, at most one object o's state can be changed from s_o to s'_o (s_o is* **null** *if st creates o).*

Proof. INST creates a single object. SET changes one single object's state. No other rules change object state. □

Definition 3 (Computation Path). *A computation path p is a finite sequence of steps $st_1 \ldots st_i$ such that $st_1 st_2 \ldots st_{i-1} st_i$ $= (S_0, r_1, S_1)(S_1, r_2, S_2) \ldots (S_{i-2}, r_{i-1}, S_{i-1})(S_{i-1}, r_i, S_i)$.*

Here we only consider finite paths as is common in process algebra, which simplifies our presentation. Infinite paths can be interpreted as a set of ever-increasing finite paths.

Definition 4 (Observable Behavior). *The observable behavior of a computation path p, $ob(p)$, is the label sequence formed by appending all of the labels occurring on the (nonlocal) steps in p.*

Note that this definition encompasses I/O behavior elegantly since the nonlocal messages are observables. I/O in KernelCoqa can be viewed as a fixed object which is synchronously or asynchronously sent nonlocal (and thus observable) messages. For technical reasons we also include

task end steps as observables which they are intuitively not, so we prove a somewhat stronger result than is necessary.

Definition 5 (Observable Equivalence)**.** *Two paths p_1 and p_2 are observably equivalent iff $ob(p_1) = ob(p_2)$, written $p_1 \equiv p_2$.*

Definition 6 (Object-blocked)**.** *A task t is in an* object-blocked state *S at some point in a path p if it* would *be enabled for a next step $st = (S, r, S')$ for which r is a* GET *or* SET *step on object o,* except *for the fact that there is a capture violation on o: one of the $R \subseteq$ or $W \subseteq$ preconditions of the* GET/SET *fails to hold in S and so the step cannot in fact be the next step at that point.*

Definition 7 (Task Sub-path)**.** *Given a fixed p, for some t a* sub-path *sp_t of p is a sequence of steps in p which are all local steps of task t. A* maximal sub-path *is a sp_t in p which is longest: no local t steps in p can be added to the beginning or the end of sp_t to obtain a longer sub-path.*

Note that the steps in sp_t need not be adjacent in p, they can be interleaved with steps belonging to other tasks.

Definition 8 (Pointed Maximal Sub-path)**.** *For a given path, a* pointed maximal sub-path *for t ($pmsp_t$) is a maximal sub-path sp_t with either 1) it has one nonlocal step appended to its end or 2) there are no more t steps ever in the path.*

The second case is the technical case of when the (finite) path has ended but the task t is still running. The last step of a $pmsp_t$ is called its *point*. We omit the t subscript on $pmsp_t$ when we do not care which task a *pmsp* belongs to.

Since we have extended the *pmsp* maximally and have allowed inclusion of one nonlocal step at the end, we have captured all the steps of any path in some *pmsp*:

Fact 1. *For a given path p, all the steps of p can be partitioned into a set of pmsp's where each step of p occurs in precisely one pmsp.*

Given this fact, we can make the following unambiguous definition.

Definition 9 (Indexed *pmsp*)**.** *For some fixed path p, define $pmsp_{t,i}$ to be the ith pointed maximal sub-path of task t in p, where all the steps of the $pmsp_{t,i}$ occur after any of $pmsp_{t,i+1}$ and before any of $pmsp_{t,i-1}$.*

The $pmsp_{t,i}$ are the units which we need to serialize: they are all spread out in the initial path p, and we need to show there is an equivalent path where each *pmsp* runs in turn as an atomic unit.

Definition 10 (Path around a $pmsp_{t,i}$)**.** *The* path around a *$pmsp_{t,i}$ is a finite sequence of all of the steps in p from the first step after $pmsp_{t,i-1}$ to the end of $pmsp_{t,i}$ inclusive. It includes all steps of $pmsp_{t,i}$ and also all the interleaved steps of other tasks.*

Definition 11 (Waits-for and Deadlocking Path)**.** *For some path p, $pmsp_{t_1,i}$* waits-for *$pmsp_{t_2,j}$ if t_1 goes into a object-blocked state in $pmsp_{t_1,i}$ on an object captured by t_2 in the blocked state. A deadlocking path p is a path where this waits-for relation has a cycle: $pmsp_{t_1,i}$ waits-for $pmsp_{t_2,j}$ while $pmsp_{t_2,i'}$ waits-for $pmsp_{t_1,j'}$.*

Hereafter we assume in this theoretical development that there are no such cycles. In Coqa deadlock is an error that should have not been programmed to begin with, and so deadlocking programs are not ones we want to prove facts about.

Definition 12 (Quantized Sub-path and Quantized Path)**.** *A* quantized sub-path *contained in p is a $pmsp_t$ of p where all steps of $pmsp_t$ are adjacent in p. A* quantized path *p is a path consisting of a sequence of quantized sub-paths.*

The main technical Lemma is the following Bubble-Down Lemma, which shows how local steps can be pushed down in the computation. Use of such a Lemma is the standard technique to show atomicity properties. Lipton [16] first described such a theory, called *reduction*; his theory was later refined by [15]. In this approach, all state transition steps of a potentially interleaving execution are categorized based on their commutativity with adjacent steps: a right mover, a left mover, a both mover or a non-mover. The reduction is defined as moving the transition steps in the allowed direction. The theory was later formulated as a type system in [9] to verify whether Java code is atomic. In our case, we show the local steps are right movers; in fact they are both-movers but that stronger result is not needed.

Lemma 2 (Bubble-down Lemma)**.** *For any path with any two adjacent steps st_{r_1}, st_{r_2} where st_{r_1} is in $pmsp_{t_1,i_1}$ and st_{r_2} is in $pmsp_{t_2,i_2}$ for $t_1 \neq t_2$, if it is not the case that $pmsp_{t_1,i_1}$ waits-for $pmsp_{t_2,i_2}$ and if st_{r_1} is a local step, then swapping the steps st_{r_1} and st_{r_2} by swapping the order of applying r_1 and r_2 we can get an observably equivalent computation path. More precisely, this equivalent path starts in the same state that st_{r_1} started in, and ends in the same state st_{r_2} ends in, but has the rules r_1 and r_2 swapped.*

See [18] for the proof of this Lemma and the Theorems below that follow from it. Given this Lemma we can now directly prove the Quantized Atomicity Theorem.

Theorem 1 (Quantized Atomicity)**.** *For all paths p there exists an observably equivalent path p' that is quantized.*

We can also show that KernelCoqa is free of data races, and large zones of mutual exclusion on objects are obtained in tasks.

Theorem 2 (Data Race Freedom)**.** *For all paths, no two different tasks can access a field of an object in adjacent steps, where at least one of the two accesses changes the value of the field.*

Theorem 3 (Mutual Exclusion over Tasks)**.** *It can* never *be the case that two tasks t_1 and t_2 overlap execution in a sequence of steps $st_1 \ldots st_n$ in a path (i.e. nether task begin*

or end of $t_{1/2}$ occurs in these steps, and $t_{1/2}$ both do take some steps in this interval; other tasks could also step as well here), and in those steps both t_1 and t_2 write the same object o, or one reads while the other writes the same object.

Mutual exclusion over tasks is a strong notion of mutual exclusion in terms of the span of the zone of mutual exclusion – it holds over the lifetime time of the whole task. Java's `synchronized` provides mutual exclusion on an object, but it is shallow in the sense that it only spans the code in enclosed by the `synchronized` method/block, and not the methods that code may invoke.

4. Implementation

```
class BankMain extends CObject{
  public void main(String [] args)  {
     ...
     //Translator-generated code for bank->(...)
     TransferWrap _v_1 = new TransferWrap();//(A1)
     _v_1.setTarget(bank);
     _v_1.setArg0("Alice");
     _v_1.setArg1("Bob");
     _v_1.setArg2(300);
     Task.rootExecutor.submit(_v_1);        //(A2)
     ...
  }
  ...
  // Translator generated inner class
  class TransferWrap implements Runnable {
     Bank target;
     String arg0, arg1;
     int arg2;
     void setTarget(Bank target){ this.target = target;}
     void setArg0(String arg0)  { this.arg0 = arg0;    }
     ...
     public void run(){
        Task t = new Task();                  //(A3)
        target.transfer(t,arg0,arg1,arg2);
        t.finish();                           //(A4)
     }
  }
}
```

Figure 11. Translation of the fragment `bank->transfer()`

We have implemented a prototype of Coqa, called CoqaJava, as a Java extension. Polyglot [25] was used to construct a translator from CoqaJava to Java so that nearly all of the existing Java syntax and type system can be reused.

All language features introduced in Fig. 8 are included in the prototype. A CoqaJava program is much like a Java program: the `o.m(arg)` Java syntax is a local message send when the same syntax is viewed as a CoqaJava program. The main difference is how CoqaJava includes nonlocal synchronous/asynchronous message sending operators ⇒ and ->, and how it discards Java's threading model in which threads are created and manipulated explicitly via objects of the `java.lang.Thread` class. Instead, parallel computations are created via asynchronous messaging and concurrently running tasks are coordinated via the Coqa mechanisms as described in the previous sections. Consequently, CoqaJava programmers do not need Java's `synchronized` method or block to coordinate threaded computations. Note we do not claim that CoqaJava offers the final set of task and coordination primitives, just the core ones. There is clearly need for higher-level coordination and that is planned as future work.

```
class Bank extends CObject {
  ...
  public void transfer(Task t, String from, String to, int b){
     ...
     Account afrom =
         (Account)htable. get_del(t, from);//(B1)
     ...
  }
}
class HashTable extends CObject {
  ...
  // Translato-generated delegate method
  Object get_del(Task t, String arg0) {
    Task subTask = new Task();            //(B2)
    Task.record(subTask, t);              //(B3)
    Object ret = get(subTask, arg0);
    subTask.finish();                     //(B4)
    return ret;
  }
}
```

Figure 12. Translation of the fragment `htable⇒get()`

Translation Overview In the implementation, we introduce a new Java base class called `CObject`. This class is invisible to programmers, and the CoqaJava translator converts all user classes to be subclasses of this base class.

`CObject` implements the algorithm for checking and updating an object's capture sets, which is activated via invoking `readReq()` or `writeReq()` provided by the `CObject`. The CoqaJava translator enforces that the two methods are invoked whenever an object field is accessed by a task.

`Task` is another Java base class that CoqaJava programmers do not directly see. A `Task` object represents a task or a subtask created by ⇒ or ->. The translator inserts code for creating a new `Task` object before a (sub)task starts to run. The `Task` object then coordinates the object capture on behalf of the running task and records captured objects so that they can be freed when the task finishes. Fig. 11 shows by way of example how the translator maps an asynchronous message send `bank->transfer()` of the earlier example in Fig. 1: it is wrapped up as instantiation of a translator-generated inner class `TransferWrap` (line `A1`), in which a new `Task` object is created (line `A3`). Similarly, Fig. 12 gives an example of translation for a synchronous message send `htable⇒get()` of the earlier example in Fig. 5: it is converted to a call to a translator-generated delegate method `get_del()` (line `B1`) in the `HashTable` class, where a subtask object is instantiated (line `B2`) and the relationship between the subtask and its creator is recorded (line `B3`). When

CoqaJava Code	Java Code
class $tn\{\ldots\}$	**class** tn **extends CObject**$\{\ldots\}$
$e.mn(e')$	$e.mn(\texttt{t}, e')$
$\tau\ mn(\tau\ x)\ \{e\}$	$\tau\ mn(\texttt{Task t}, \tau\ x)\ \{e\}$
$\tau\ getter()\ \{e\}$	$\tau\ getter(\texttt{Task t})\ \{\textbf{this}.\texttt{readReq}(\texttt{t}); e\}$
$setter(\tau\ e)\ \{e'\}$	$setter(\texttt{Task t}, \tau\ e)\ \{\textbf{this}.\texttt{writeReq}(\texttt{t}); e'\}$
`exclusive` label	a **CObject** field

$e \rightarrow mn(e')$	`Task t = new MnWrap();`
	`t.setTarget(e);`
	`t.setArg(e');`
	`Task.rootExecutor.submit(t);`
	
	`//Translator-generated inner`
	`//class for` τ_e
	class *Mn*`Wrap` **implements** `Runnable{`
	`Task t;`
	τ_e `target;`
	$\tau_{e'}$ `arg;`
	void `setTarget(`τ_e `target)`
	`{ this.target = target; }`
	void `setArg(`$\tau_{e'}$ `arg)`
	`{ this.arg = arg; }`
	public void `run()`
	`{ t = new Task();`
	`target.`*mn*`(t, arg);`
	`t.finish(); }`
	`}`

$e \Rightarrow mn(e')$	$e.mn$`_del(t,` e'`)`
	
	`//Translator-generated method for` τ_e
	τ *mn*`_del(Task t,` $\tau_{e'}$ `arg)`
	`{ Task nt = new Task();`
	`Task.record(nt, t);`
	τ_e `ret =` *mn*`(nt, arg);`
	`nt.finish();`
	return `ret; }`

Figure 13. CoqaJava to Java Syntax-Directed Translation

a (sub)task finishes, a call to `finish()` is invoked on its representative `Task` object (Fig. 11 line `A4` and Fig. 12 line `B4`). The finishing process notifies all objects captured by the ending task so that these objects can remove the task from their capture sets.

We utilize the new `java.util.concurrent` package in our prototype implementation. Every CoqaJava application has a built-in `ScheduledThreadPoolExecutor`, named `rootExecutor`. It serves as an execution engine for the tasks of the application. Fig. 11 line `A2` shows how a task is submitted to the executor for execution. Subtasks can always be running in the parent thread in CoqaJava because their computations do not overlap.

Fig. 13 gives the complete code translation schema describing how CoqaJava programs are mapped to Java. The remaining Java, including the primitive datatypes, `public` and `protected` modifiers *etc.*, that are not shown in this table are generally kept unchanged. We currently leave some Java language features out of our implementation, including inner classes, native methods, field uses which are not via accessors, exceptions, and static fields; these features are not difficult to include.

As shown in Fig. 13, every method is translated to carry an extra parameter, `Task t`, that records which task the invocation is in. In CoqaJava, objects are restricted to have only `private` fields which are accessed by getters/setters; this restriction is in fact only there for implementation simplicity and will be lifted in the near future. The CoqaJava translator translates all getters/setters to capture an object on behalf of a task before it accesses a field by calling `readReq()`/`writeReq()` on the object. For instance, if a task reads an object field via its getter method, the first thing the getter method does is to invoke `readReq()` provided by `CObject` on the current object. If the read request is granted, `readReq()` updates the capture sets of the object, then returns silently. If the read is not allowed at this moment, the thread running the task will be put into a wait state on the `readReq`. The waiting task will be notified when the requested object becomes available, then its read request can be fulfilled and its computation resumes. The "exclusive" label declared in a CoqaJava class is translated to a boolean value in an instance field of `CObject` which will be checked first whenever a read request is made by a task.

5. Related Work

Atomicity via STM The property of atomicity is commonly discussed in the context of STM systems: systems with optimistic and concurrent execution of code that should be atomic, and the ability to rollback when contention happens. Programming language support for STM started with Harris and Fraser [12], and then appeared in many languages and language extensions, such as Transactional Monitors [30] for Java, Concurrent Haskell [13], AtomCaml [26], Atomos [7], Fortress [3] and Chapel [8]. We have compared the STM-approach with our work in Sec. 2.3.

Open nesting as a language feature is studied in [24], where open nesting is linked to a notion of *abstract serializability*: a concurrent execution of a set of transactions is viewed as abstractly serializable if the resulting abstract view of data is consistent with some serial execution of those transactions. It remains to be seen how open nesting can be rigorously stated or proved as a property dependent on the semantics of data. Quantized atomicity however clearly defines what is preserved and what is broken, regardless of the program semantics. A limited open nesting mechanism is provided by Atomos. See Sec. 2.2.

Atomicity in Non-STM Languages Of the non-STM languages, our work is most related to actor languages. Actors [1, 2] provide a simple concurrent model where each actor is a concurrent unit with its own local state. Inter-actor communication is achieved by asynchronous messaging. Full atomicity is preserved because executing each actor method does not depend on the state of other actors and each method execution is trivially serializable. Actors are also deadlock free. Actors have influenced the design of Erlang [4], the E language [23], and Scala [11], amongst others.

Actors however are a model more suited to loosely-coupled distributed programming. For tightly-coupled message sequences, programming them in the pure Actor model means breaking off each method after each send and wrapping up the continuation as a new actor method. Typically when Actor languages are implemented, additional language constructs are included to ease programmability, but there is still a gap. Consider [11] as a representative of this category of languages. Asynchronous messaging is provided (via the `o!m(e)` syntax), and at the same time one can use expression `receive`$\{p\}$ to wait for messages where p is a pattern matching of messages. The latter construct indeed makes declaring a method for each return value unnecessary. Consider a programmer intends to model the following Java code:

```
x = o.m1(v);
y = o.m2(x);
...
```

the typical code snippet in [11] would look like:

```
o!m1(v)
receive {
   case x => o!m2(x);
             receive {
                case y: ...
             }
}
```

Note that by doing such a transformation, a few invariants which held in Java are now not preserved. For instance, when the first `receive` matches the result of `x`, `x` might not be a return value from the actor `o`. Due to concurrency, such a message in fact might come from any actor. In addition, the `receive` construct is required to never return normally to the enclosing actor (no statement should be following a `receive` block). This requires programmers to wrap up all continuations in the `receiver` blocks. If a Java program is meant to have 10 method invocations, the resulting actor-based program would have to have nested `receive` blocks of depth 9. Lastly, the introduction of `receive` is essentially a synchronization point. It will make the model loses the theoretical properties the pure Actor model has, atomicity and deadlock freedom.

Argus [17] pioneered the study of atomicity in object-oriented languages. Like actors it is focused on loosely coupled computations in a distributed context, so it is quite removed in purpose from Coqa but there is still overlap in some dimensions. Argus is a transaction-based system in that actions can rollback, but unlike STM systems they use a combination of locking and rollback to enforce this. Argus objects are local except for special objects called *guardians*, placed one per distributed site. Only invocation of guardian methods can lead to parallel computation (called *atomic actions*), while in our language, an asynchronous invocation to any object can lead to parallel computation. Argus allows nested transactions, called *subactions*. Unlike our subtasking, when a subaction ends, all its objects are merged with the parent action, instead of being released early to promote parallelism as a subtask does.

Guava [5] was designed with the same philosophy as Coqa: code is concurrency-aware by default. The property Guava enforces is race freedom, which is a weaker and more low-level property than the quantized atomicity of Coqa. JShield [21] is a similar design in spirit. In [28], a data-centric approach is described for programmers to express their needs for data race and atomicity. Programmers add annotations to Java programs to express how different fields or method parameters should be accessed.

6. Conclusion and Future Work

Coqa is a foundational study of how concurrency can be built deeply into object models; in particular our target is tightly coupled computations running concurrently on multi-core CPUs. Coqa has a simple and sound foundation – it is defined via only three forms of messaging, which account for (normal) local message send, thread spawning via asynchronous message send, and atomic subtasking via synchronous nonlocal send. We formalized Coqa as the language KernelCoqa, and proved that it observes a wide range of good concurrency properties, in particular *quantized atomicity*: each and every method execution can be factored into just a few quantum regions which are each atomic. We justify our approach by implementing CoqaJava, a Java extension incorporating all of the Coqa features.

In the future, we intend to make our translator more efficient and language more expressive:

- While CoqaJava implements most of existing Java syntax, a few features were left out for simplicity which we plan on adding shortly. These features include static fields, implicit getters/setters, inner classes, and exceptions.
- We also intend to optimize CoqaJava. For instance static analysis such as various lock inference techniques [14] can provide useful insights on finding mutual exclusion region statically so that objects in a mutual exclusion region does not require to be captured.
- We intend to add more language features such as synchronization constraints [1, 6, 22].

A detailed plan of these topics are given [18].

References

[1] AGHA, G. *ACTORS : A model of Concurrent computations in Distributed Systems*. MITP, Cambridge, Mass., 1990.

[2] AGHA, G., MASON, I. A., SMITH, S. F., AND TALCOTT, C. L. A foundation for actor computation. *Journal of Functional Programming 7*, 1 (1997), 1–72.

[3] ALLEN, E., CHASE, D., LUCHANGCO, V., RYU, J. W. M. S., STEELE, G., AND TOBIN-HOCHSTADT, S. The Fortress Language Specification (Version 0.618), April 2005.

[4] ARMSTRONG, J. Erlang — a Survey of the Language and its Industrial Applications. In *INAP'96 — The 9th Exhibitions and Symposium on Industrial Applications of Prolog* (Hino, Tokyo, Japan, 1996), pp. 16–18.

[5] BACON, D. F., STROM, R. E., AND TARAFDAR, A. Guava: a dialect of java without data races. In *OOPSLA '00* (New York, NY, USA, 2000), ACM Press, pp. 382–400.

[6] BENTON, N., CARDELLI, L., AND FOURNET, C. Modern concurrency abstractions for c#. *ACM Trans. Program. Lang. Syst. 26*, 5 (2004), 769–804.

[7] CARLSTROM, B., MCDONALD, A., CHAFI, H., CHUNG, J., MINH, C., KOZYRAKIS, C., AND OLUKOTUN, K. The atomos transactional programming language. In *PLDI'06* (Ottawa, Ontario, Canada, June 2006).

[8] CRAY INC. Chapel Specification, 2005.

[9] FLANAGAN, C., AND QADEER, S. A type and effect system for atomicity. In *PLDI '03* (New York, NY, USA, 2003), ACM Press, pp. 338–349.

[10] GRAY, J. Notes on data base operating systems. In *Operating Systems, An Advanced Course* (London, UK, 1978), Springer-Verlag, pp. 393–481.

[11] HALLER, P., AND ODERSKY, M. Event-based programming without inversion of control. In *Proc. Joint Modular Languages Conference* (2006), Springer LNCS.

[12] HARRIS, T., AND FRASER, K. Language support for lightweight transactions. In *OOPSLA'03* (2003), pp. 388–402.

[13] HARRIS, T., HERLIHY, M., MARLOW, S., AND PEYTON-JONES, S. Composable memory transactions. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming, to appear* (Jun 2005).

[14] HICKS, M., FOSTER, J. S., AND PRATIKAKIS, P. Lock inference for atomic sections. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)* (2006).

[15] LAMPORT, L., AND SCHNEIDER, F. B. Pretending atomicity. Tech. Rep. TR89-1005, Digital Equipment Corporation, 1989.

[16] LIPTON, R. J. Reduction: a method of proving properties of parallel programs. *Commun. ACM 18*, 12 (1975), 717–721.

[17] LISKOV, B. Distributed programming in argus. *Commun. ACM 31*, 3 (1988), 300–312.

[18] LIU, Y. D., LU, X., AND SMITH, S. F. Coqa: Concurrent objects with quantized atomicity (long version), March 2007. Available at `http://www.cs.jhu.edu/~xiaoqilu/Concurrency/`.

[19] LOMET, D. B. Process structuring, synchronization, and recovery using atomic actions. *SIGOPS Oper. Syst. Rev. 11*, 2 (1977), 128–137.

[20] MANSON, J., PUGH, W., AND ADVE, S. V. The java memory model. In *POPL '05* (New York, NY, USA, 2005), ACM Press, pp. 378–391.

[21] MATEU, L. A java dialect free of data races and without annotations. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing* (Washington, DC, USA, 2003), IEEE Computer Society, p. 52.2.

[22] MATSUOKA, S., AND YONEZAWA, A. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions in Concurrent Object-Oriented Programming*, G. Agha, P. Wegner, and A. Yonezawa, Eds. MIT Press, 1993, pp. 107–150.

[23] MILLER, M. The E Language, `http://www.erights.org`.

[24] NI, Y., MENON, V., ADL-TABATABAI, A.-R., HOSKING, A. L., HUDSON, R. L., MOSS, J. E. B., SAHA, B., AND SHPEISMAN, T. Open nesting in software transactional memory. In *ACM SIGPLAN 2007 Symposium on Principles and Practice of Parallel Programming* (March 2007).

[25] NYSTROM, N., CLARKSON, M. R., AND MYERS, A. C. Polyglot: An extensible compiler framework for java. In *Compiler Construction: 12'th International Conference, CC 2003* (NY, Apr. 2003), vol. 2622, Springer-Verlag, pp. 138–152.

[26] RINGENBURG, M. F., AND GROSSMAN, D. AtomCaml: First-class atomicity via rollback. In *ICFP'05* (September 2005), pp. 92–104.

[27] SINGHAL, M. Deadlock detection in distributed systems. *IEEE Computer 22*, 11 (1989), 37–48.

[28] VAZIRI, M., TIP, F., AND DOLBY, J. Associating synchronization constraints with data in an object-oriented language. In *POPL '06* (New York, NY, USA, 2006), ACM Press, pp. 334–345.

[29] WELC, A., HOSKING, A. L., AND JAGANNATHAN, S. Transparently reconciling transactions with locking for java synchronization. In *ECOOP'06* (2006), pp. 148–173.

[30] WELC, A., JAGANNATHAN, S., AND HOSKING, A. L. Transactional monitors for concurrent objects. In *ECOOP'04* (2004), vol. 3086 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 519–542.