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 the desirable behaviors of atomicity, mutual exclusion, and race freedom automatically built in. A key property of our model is the notion of quantized atomicity: every concurrent program execution can be viewed as being divided into quantum regions of atomic execution, greatly reducing the number of interleavings to consider. So rather than building atomicity locally, with small declared zones, we build it globally, down from the top. 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

Coqa (for *C*oncurrent *o*bjects with *q*uantized *a*tomicity) is a new object-oriented language aimed at facilitating programming in a multi-core CPU environment. Programming multi-core CPUs requires much greater programmer skill, and is one of the most significant new demands programmers will face in the coming decade. The design goal of Coqa is to build a language in which it is easier to naturally write concurrent programs with good concurrency properties. Unlike Java where good properties such as race freedom can only be achieved if the programmer *explicitly* declares it by using `synchronized`, the "default" mode in Coqa is inverted: good properties of race freedom, mutual exclusion, and atomicity are preserved unless programmers explicitly declare otherwise.

Existing concurrent object language designs are numerous and include for example [Agh90,Arm96,Mil,BST00]. What makes our work novel is the intrisic properties Coqa preserves. Most important is 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. With tightly coupled computation running on multi-core CPUs, data sharing between threads is very common and the patterns are more complex than a single-core CPU due to random variations in scheduling. To support atomicity, Coqa takes the route of "atomicity-by-design" for each method: atomicity is *ubiquitous* because by default each complete method execution is observably atomic. Note this is much stronger than the `synchronized` methods

of Java: the Coqa method *and all methods it invokes* are viewed as happening atomically. The `synchronized` methods in Java only provide a shallow notion of mutual exclusion.

One particular challenge of whole-method atomicity is that it can be overly strong, and the resulting executions will not be efficient, or may even deadlock if there is significant contention across methods. For this reason, Coqa allows programmers to relax whole-method atomicity by dividing a method into a small number of discrete zones of atomicity (called *quanta* in Coqa), and each quantum is serializable regardless of the interleaving of the actual execution. This property, called *quantized atomicity*, is preserved for all Coqa programs. The main appeal is to significantly reduce the number of interleavings possible in concurrent program runs, and thus to ease the debugging burden. If two pieces of code each have 100 execution steps, reasoning tools would have to consider 101^{100} interleaving scenarios; however, if the aforementioned 100 steps can be split into 3 atomic quanta, there are only 4^3 possibilities to consider. Actors [Agh90,AMST97] were in some sense the starting point for the design of Coqa: atomicity is preserved for each Actor method because its execution once initiated does not depend on the state of other actors and each method is therefore trivially serializable. Actors' ubiquitous atomicity arises from the fact that the model supports only asynchronous messaging, and so methods once initiated cannot receive outside inputs.

Another design goal of Coqa is to make a concurrent language design that naturally meshes well with object-oriented language features. This stands in constrast to the non-object-based syntax and semantics commonly used in existing languages for concurrent programming. Language abstractions such as library class `Thread`, thread spawning via its `start` method and `synchronized` blocks in Java, and the `atomic` blocks in various Software Transactional Memory (STM) systems that have been adopted into OO languages [CMC+06], are not that different from what was used three decades ago in non-object-oriented languages [Lom77].

Existing language models fall short of achieving the goals of both ubiquitous atomicity and easy OO-style concurrent programmming. Ubiquitous atomicity is a global property of all programs; Java does not have a notion of atomicity built into the langauge and the form of atomicity in STM systems is only local atomicity. STM systems also require rollbacks to deal with atomicity-breaking contentions and are known to be inapplicable to I/O-intensive applications, such as GUI and network systems, so they can never be ubiquitous. Out of the desire of pervasiveness, we take a blocking and not a rollback approach to achieve atomicity. The Actor model achieves ubiquitous atomicity, but programming in Actors is very different from what programmers are used to, since with pure asynchronous messaging any processing of a message reply must be handled by a completely new message, necessarily chopping up methods into many small pieces. So, Coqa shares the spirit of ubiquitous atomicity of Actors, but allows more familiar synchronous messaging syntax to be used which avoids the need to break up methods. to have language level

messaging	what it is	why you should use it
`o .m(v)`	intra-task messaging	promotes mutual exclusion and atomicity
`o -> m(v)`	task creation	promotes parallelism by starting up a new task
`o => m(v)`	sub-tasking	promotes parallelism by encouraging early free

Fig. 1. The Three Messaging Mechanisms and Their Relative Strengths

In this paper, we formalize Coqa in a formal system called KernelCoqa, in which we prove the properties of quantized atomicity, mutual exclusion and race freedom. We have also implemented a prototype language CoqaJava as a Java extension which simply replaces Java threads with our new forms of object messaging.

2 Informal Overview

The concurrency unit in our language is a *task*. A task is a unit of execution that can potentially be interleaved with other units. Tasks are closely related to (logical) threads, but come with inherent atomicity properties not found in threads, and we coin a new term to reflect this distinction. Coqa has a very simple syntax: the only difference from the Java object model is a richer syntax to support object messaging, as summarized in Fig. 1. Beyond the familiar `o .m(v)` message send expression, `o -> m(v)` and `o => m(v)` are additionally provided for *task creation* (a form of thread spawning) and *subtasking* (a form of thread open nesting), respectively.

The Running Example Throughout the section, we will use a simple example of basic banking operations, including account opening and balance transfer operations, as shown in Fig. 2. Bank accounts are stored in a hash table, implemented in a standard manner with bucket lists.

2.1 Task Creation

Tasks are created by simply sending asynchronous messages to objects, using the `o -> m(v)` expression. This is a more "object-based" thread creation than the current practice in Java, where a special `Thread` class is used. This notion is more aligned with Actor languages, where all message passings can be viewed as thread creations. In Fig. 2, the top-level `main` method starts up three concurrent tasks, two balance transfers and one account open, by the invocations of lines `M1`, `M2` and `M3`. Syntax `bk->transfer("Alice", "Bob", 3)` indicates an asynchronous message `transfer` sent to object `bk` 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. To keep the language simple, asynchronous invocations in Coqa do not return values.

```
class BankMain {
  public static void main (String [] args)  {
     Bank bk = new Bank();
     bk.open("Alice", 10); bk.open("Bob", 20); bk.open("Cathy", 30);
     bk->transfer("Alice", "Bob", 3);             //(M1)
     bk->transfer("Cathy", "Alice", 5);           //(M2)
     bk->open("Dan", 40);                         //(M3)
}}
class Bank {
  void transfer (String from, String to, int bal) {
     Status status = new Status();               //(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)
  }
  void open(String n, int b) { htable.put(n, new Account(n, b));}
  private HashTable htable = new HashTable();
}
class Account  {
  Account(String n, int b) {name = n; bal = b; }
  void deposit(int b, Status s) { bal += b; s.append("+", b, name); }
  void withdraw(int b, Status s) {bal -= b;  s.append("-", b, name); }
  private String name;
  private int bal;
}
class Status  {
  void append(String s, int i, String name)  {info.append(s + i + name);}
  private StringBuffer info = new StringBuffer();
}
```

Fig. 2. A Banking Program

2.2 Intra-Task Messaging

Message send `o .m(v)` is the same syntax as Java, but has different semantics giving stronger atomicity properties: when invoked, object `o` will be *captured* by the invoking task and cannot be used by other tasks until the current task is complete. Capturing is a blocking mechanism, but unlike Java where programmers need to explicitly specify what to lock and when to lock, the capture and blocking of objects is built into Coqa.

This intuitive definition for `o .m(v)` is the programmer view, but is not an efficient implementation strategy: only mutation affects the preservation of atomicity, and so we actually only need to capture objects "lazily" when their fields are read and written. Our notion of "capture" is a standard two-phase

non-exclusive read lock and exclusive write lock [Gra78]. 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 task. Two-phase locking optimizes our model since reads are overwhelmingly more common than writes in most programs. Many other optimizations are also possible by static analysis, a topic we leave to future work.

The preservation of atomicity can be seen in the `transfer` method of Fig. 2: the `HashTable` object referenced by `htable` is captured by a task, say the task created at Line M1, and will not be released until the end of the method (and hence, the task). Therefore it is not possible for one `transfer` task to be reading from the `HashTable` object while at the same time a different `transfer` task is writing to it.

2.3 Subtasking

The model we have presented thus far admits significant parallelism if most object accesses are read accesses. Blocking is possible, however, when frequent writes are needed. For instance, consider the parallel execution of the two tasks spawned by `(M1)` and `(M3)`. One of them will be blocked as `(M1)` reads from the `HashTable` object, while `(M3)` attempts to write.[1] And the task being blocked cannot make any progress until the other task completes and release its captured object. Intuitively, the task of adding `Dan` as a new account, `(M3)`, is totally unrelated to the task of transferring money from `Alice` to `Bob`, `(M1)`, except for their shared access to the `HashTable` object. There should be at least some parallelism possible between the two tasks.

Coqa achieves this by allowing programmers to spawn off the access of the `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 captured objects used to achieve this subtask can be "freed", *i.e.* no longer considered captured. In terms of syntax, the only change to the source code of `transfer` in Fig. 2 is to change the dot (`.`) messagings at `(A2)` and `(A4)` to `⇒` for subtask creation messaging. In this case, the task t created at `(M1)` spawns a subtask t' at `(A2)` via `⇒`. The `Hashtable` object will be captured by t' but not t. More parallelism is achieved by such subtasking: other tasks waiting to capture the `HashTable` object would have to block for the duration of t instead of the much shorter span of t' if (.) was used. Subtasking is a synchronous invocation, *i.e.*, the task executing `transfer` waits until its subtask executing `get` returns a result. But the subtask has a *distinct* capture set of its own. And like a task, a subtask frees objects in its capture set when it finishes.

[1] Strictly speaking, the read-write conflict happens on the object representing the bucket list inside the `HashTable`, but we omit this detail since we do not have space to give the source code for the internals of the `HashTable`.

A subtask is also a task, so it prevents arbitrary interleaving. The change at line `(A2)` from (.) to `⇒` admits interleaving between task `(M1)` and `(M3)` that was not allowed before, but it does not mean that arbitrary interleaving can occur; for example, if `M1` were 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 related to open nesting in STM systems [NMAT+07,CMC+06]. 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 [CMC+06], 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, "Can a subtask access objects already captured by its enclosing task(s)?"

We could in theory follow Atomos' approach. 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.

We believe the essence of having a subtasking relationship between a parent and a child is that the parent should generously share its resources with the child. Therefore accessing the `Status` in the subtask is perfectly legal in Coqa. Observe that the relationship between a task and its subtask is synchronous, so there is no concern of interleaving between a task and its subtask.

2.4 Properties

Quantized Atomicity 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. The main reason why a programmer wants to declare a subtask is to open a communication channel with other tasks for such sharing, as was illustrated in the subtasking example above. With subtasking, objects captured by the subtask can serve as communication points between different tasks. This is because the objects freed at the end of one subtask might be recaptured later, and the object may have been mutated by the original subtask.

Quantized atomicity is the property that for any task, its execution sequence can be viewed as a sequence of atomic regions, the *atomic quanta*, demarcated by task and subtask creation points. This atomicity property is weaker than a whole task being atomic, but as long as full task atomicity is broken only when it is really necessary (that is, a minimal number of `⇒` and `->` messagings are used),

the atomic quanta will each be large, and significant reduction of interleaving can be achieved. In reality, what matters is not that the entire method must be atomic, but that the method admits a drastically limited number of interleaving scenarios. Quantized atomicity aims to strikes a balance between what is realistic and what is reasonable.

Selective Mutual Exclusion For objects accessed by synchronous messaging, the property of mutual exclusion over mutation spans the lifetime of the current task, even *across the boundaries of quanta*. For instance, over the entire duration of any task executing `transfer` in Fig. 2, the `Status` object is guaranteed not to be mutated by any other task before the current `transfer` ends, even if other tasks have reference to `Status`. Our notion of object mutual exclusion is much stronger than what Java's `synchronized` provides: Java only guarantees the object with the method is itself not mutated by other threads[2], while we are guaranteeing the property for *all* objects which are directly or indirectly sent synchronous messages to at run time by the method, many of which may be unknown to the caller.

Race Freedom In Coqa, we show that two tasks cannot race to access any object field, except in the case where both may have been only reading from the same object field.

3 Formalization

In this section we present KernelCoqa, a small formal kernel language of Coqa.

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 $\emptyset$ as empty set. $\overrightarrow{x_n \mapsto y_n}$ denotes 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, $\mathsf{dom}(H)$. 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. 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 no effect.

KernelCoqa is an idealized object-based language with objects, messaging, and fields. Its abstract syntax is shown on the left of Fig. 3. A program P is 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 features are left out, including types and constructors. Besides local method invocations via dot (`.`) notation, synchronous and asynchronous messages are sent to objects using `=>` and `->`, respectively. A class delcared `exclusive` will have its objects write captured upon any access. This label is

[2] A variant of Java's `synchronized` allows programmers to specify what objects to be accessed in a mutually exclusive manner. Programmers still have to know beforehand the objects.

$$
\begin{array}{rcl}
P & ::= & \overrightarrow{cn \mapsto \langle l; Fd; Md\rangle} \\
Fd & ::= & \overline{fn} \\
Md & ::= & \overrightarrow{mn \mapsto \lambda x.e} \\
e & ::= & \mathbf{null} \mid x \mid cst \mid \mathbf{this} \\
 & \mid & \mathbf{new}\ cn \\
 & \mid & fn \mid fn = e \\
 & \mid & e.mn(e) \\
 & \mid & e \mathbin{\texttt{->}} mn(e) \\
 & \mid & e \Rightarrow mn(e) \\
 & \mid & \mathbf{let}\ x = e\ \mathbf{in}\ e \\
l & ::= & \texttt{exclusive} \mid \epsilon \\
cst & & \textit{constant} \\
cn & & \textit{class name} \\
mn & & \textit{method name} \\
fn & & \textit{field name} \\
x & & \textit{variable name}
\end{array}
\qquad
\begin{array}{rcl}
H & ::= & \overrightarrow{o \mapsto \langle cn; R; W; F\rangle} \\
F & ::= & \overrightarrow{fn \mapsto v} \\
T & ::= & \langle t; \gamma; e\rangle \mid T \parallel T' \\
N & ::= & \overrightarrow{t \mapsto t'} \\
R, W & ::= & \overline{t} \\
\gamma & ::= & o \mid \mathbf{null} \\
v & ::= & cst \mid o \mid \mathbf{null} \\
e & ::= & v \mid \mathbf{wait}\ t \\
 & \mid & e \uparrow e \mid \ldots \\
\mathbf{E} & ::= & \bullet \mid fn = \mathbf{E} \\
 & \mid & \mathbf{E}.m(e) \mid v.m(\mathbf{E}) \\
 & \mid & \mathbf{E} \mathbin{\texttt{->}} m(e) \mid v \mathbin{\texttt{->}} m(\mathbf{E}) \\
 & \mid & \mathbf{E} \Rightarrow m(e) \mid v \Rightarrow m(\mathbf{E}) \\
 & \mid & \mathbf{let}\ x = \mathbf{E}\ \mathbf{in}\ e \\
o & & \textit{object ID} \\
t & & \textit{task ID}
\end{array}
$$

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

Fig. 3. Language Abstract Syntax and Dyanimc Data Structure

useful for eliminating deadlocks inherent in a two-phase locking strategy, such as when two tasks first read capture an object, then both try to write capture the same object and thus deadlock.

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 the object heap, N a task ancestry mapping, and T a set of parallel tasks. Every task has a local evaluation context $\mathbf{E}$. The relevant definitions are given in Fig. 3. H is a mapping from objects o to field records tagged with their class name cn. In addition, each o has capture sets, R and W, for recording tasks that have read or write captured this object. 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.

The core single-step evaluation rules are presented in Fig. 4. The rules for Let, Return and other standard constructs are omitted here; see [LLS07]. The rules implicitly operate over some fixed program P. The Invoke rule for intra-task messaging is interpreted as a standard function application. The Task rule creates a new task via asynchronous messaging. The SubTask rule creates a subtask of the current task via synchronous messaging, and the parent task enters a **wait** state until the subtask returns. When a task finishes, all objects it has captured are freed; the TEnd and STEnd are rules for ending a task and a subtask, respectively. The two-phase locking capture policy is implemented in the Set and the Get rules. The optional `exclusive` modifier requires an object to be write captured in both rules. When a task cannot capture an object it

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

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

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

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

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

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

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

Fig. 4. KernelCoqa Core Operational Semantics Rules

needs, it is implicitly *object-blocked* on the object until it is entitled to capture it—the SET/GET rule cannot progress.

Atomicity Theorems Here we formally establish the informal claims about KernelCoqa: quantized atomicity, mutual exclusion of tasks, and race freedom. Proofs are provided in [LLS07]. The key Lemma is the Bubble-Down Lemma, Lemma 1, which shows that consecutive steps of a certain form in a computation path can be swapped to give an equivalent path. Then, by a series of bubblings, each quantum of steps can be bubbled to all be consecutive in an equivalent

computation 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 in a larger concurrent computation path. We prove in Theorem 1 that any computation 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 $o \notin \mathsf{dom}(H)$.*

Definition 2 (Local and Nonlocal Step). *A step $st_r = (S, r, S')$ denotes a transition $S \Rightarrow S'$ by rule r of Figure 4. st_r is a* local *step if r is one of the local rules: either* Get, Set, This, Let, Return, Inst *or* Invoke. *st_r 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 4, used as the observable.

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

When no confusion arises, we simply call it a path.

Definition 4 (Observable Behavior). *The* observable behavior *of a path p, $ob(p)$, is the sequence of labels the nonlocal steps in p.*

Note that this definition encompasses I/O behavior elegantly since I/O in Kernel-Coqa can be viewed as a fixed object which is sent nonlocal and thus observable messages.

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

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 $st_r = (S, r, S')$ 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 preconditions of the* Get/Set *fails to hold in S and so st_r cannot in fact be the next step at that point.*

Definition 7 (Sub-path and Maximual Sub-path). *Given a path 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.*

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*.

The *pmsp*'s 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 9 (Task Indexed *pmsp*). *For some fixed path p, define $pmsp_{t,i}$ to be the i^{th} 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 $pmsp_{t,i-1}$.*

Definition 10 (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 11 (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 consecutive 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 a path. Use of such a Lemma is the standard technique to show atomicity properties. Lipton [Lip75] first described such a theory, called *reduction*; his theory was later refined by [LS89].

Definition 12 (Equivalent Step Swap). *For two consecutive steps $st_{r_1}st_{r_2}$ in a path p, where $st_{r_1} \in pmsp_{t_1}$, $st_{r_2} \in pmsp_{t_2}$, $t_1 \neq t_2$ and $st_{r_1}st_{r_2} = (S, r_1, S')(S', r_2, S'')$,if the step swap of $st_{r_1}st_{r_2}$, written as $st'_{r_2}st'_{r_1}$, gives a new path p' such that $p \equiv p'$ and $st'_{r_2}st'_{r_1} = (S, r_2, S^*)(S^*, r_1, S'')$, then it is an equivalent step swap.*

Lemma 1 (Bubble-down Lemma). *For any path p with any two consecutive steps $st_{r_1}st_{r_2}$ where $st_{r_1} \in pmsp_{t_1}$,$st_{r_2} \in pmsp_{t_2}$ and $t_1 \neq t_2$, if it is not the case that $pmsp_{t_1}$ waits-for $pmsp_{t_2}$ and if st_{r_1} is a local step, then a step swap of $st_{r_1}st_{r_2}$ is an equivalent step swap.*

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

Theorem 2 (Data Race Freedom) *For all paths, no two different tasks can access a field of an object in consecutive 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 consecutive sequence of steps $st_{r_1} \ldots st_{r_n}$ in a path, 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.*

4 Discussion and Related Work

Implementation We have implemented a prototype of Coqa, called CoqaJava. Polyglot [NCM03] was used to construct a translator from CoqaJava to Java. All language features introduced in Fig. 3 are included in the prototype. The implementation dynamically enforces the object capture, freeing, and mutual exclusion semantics of Coqa. Refer to [LLS07] for more details about CoqaJava. Making our language more expressive and its implementation more efficient is a future goal. For instance, it will be interesting to add more concurrency-related language features, such as futures and synchronization constraints. Optimization techniques should also be able to minimize the amount of capture information that needs to be retained at runtime.

Deadlocks There are two forms of deadlock arising in Coqa. The first is inherent in two-phase locking, when an object is read captured by two tasks but neither task can further write capture it. The second form is a cyclically dependent deadlock. The first form of deadlock can be avoided by declaring the class to be `exclusive` (see Sec. 3). Programmers can also explicitly introduce interleaving via `=>` to break deadlock. In addition, there are many static and dynamic analysis techniques and tools to ensure deadlock freedom; for an overview, see [Sin89]. Observe that the most difficult issue for deadlock detection is that it must make sure deadlock is not possible for *all* possible interleaving scenarios. The precision of static techniques are reduced due to the combinatorial explosion of interleaving. Since all Coqa code observes quantized atomicity, stronger analysis results should be achievable over Coqa programs because interleaving has been significantly reduced.

Blocking vs. Rollback Atomicity is commonly addressed in STM systems via rollbacks; example approaches include Harris and Fraser [HF03], Transactional Monitors [WJH04] for Java, and Atomos [CMC+06]. Compared with blocking systems like ours, STM systems have the appeal of not introducing deadlocks. However, there is a counterpart to deadlock in STM systems, *livelock*, where rollbacks resulting from contention might result in further contentions and further rollbacks, *etc.* How frequently livelocks occur is typically gauged by experimental methods. In addition, rollback also may not be as easy as simply discarding the read/write set and retrying (see `AbortHandler`, *etc.* in [CMC+06] and `onAbort` *etc.* methods in [NMAT+07]). In terms of performance there have been no detailed studies that we know of comparing locking and rollback. A good overview of the pros and cons of blocking and rollback appears in [WHJ06].

A primary reason why Coqa does not take the rollback approach is our desire for ubiquitous atomicity, even for I/O-intensive applications. Existing STM systems provide atomicity guarantees only for code explicitly specified by programmers, say, by declaring a block to be `atomic`; I/O cannot occur in these regions since it cannot generally be undone. In order to make sure that the system can roll back to the state before an abandoned transaction, a STM system

needs to perform bookkeeping on the initial state of every transaction. So programmers have to be stingy in the number of atomic blocks declared, to avoid the overhead of such bookkeeping growing unexpectedly large with increasing number of threads and transaction sizes. As a result, in a large number of STM systems [HF03,WJH04,Cra05], code by default runs in a mode with no atomicity guarantees, and the interleaving of this code with atomicity-preserving code in fact can break the atomicity of the latter, an unfortunate consequence known as weak atomicity [CMC+06].

Atomicity in Non-STM Languages Of the non-STM languages, our work is most related to Actor languages. Actors [Agh90,AMST97] 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 so each method execution is trivially serializable. Actors are also deadlock free. 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 [Arm96,Mil,HO06,YBS86], additional language constructs (such as futures, and explicit continuation capture) are included to ease programmability, but there is still a gap in that the most natural mode of programming, synchronous messaging, is not fully supported, only limited forms thereof. We elect to support full synchronous messaging so that Coqa coding style can be extremely close to standard programming practice.

Argus [Lis88] 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 remote in purpose from Coqa but there is still overlap in some dimensions. 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 [BST00] 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.

5 Conclusion and Future Work

Coqa is a foundational study of how concurrency can be built deeply into object models; our particular target is tightly coupled computations running concurrently on multi-core CPUs. Coqa has a very 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*. We justify our approach by implementing CoqaJava, a Java extension incorporating all of the Coqa features.

References

[Agh90] Gul Agha. *ACTORS : A model of Concurrent computations in Distributed Systems*. MITP, Cambridge, Mass., 1990.

[AMST97] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[Arm96] J. Armstrong. Erlang — a Survey of the Language and its Industrial Applications. In *INAP'96 — The 9th Exhibitions and Symposium on Industrial Applications of Prolog*, pages 16–18, Hino, Tokyo, Japan, 1996.

[BST00] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: a dialect of java without data races. In *OOPSLA '00*, pages 382–400, New York, NY, USA, 2000. ACM Press.

[CMC+06] B. CarlStrom, A. McDonald, H. Chafi, J. Chung, C. Minh, C. Kozyrakis, and K. Olukotun. The atomos transactional programming language. In *PLDI'06*, Ottawa, Ontario, Canada, June 2006.

[Cra05] Cray Inc. Chapel Specification, 2005.

[Gra78] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag.

[HF03] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA'03*, pages 388–402, 2003.

[HO06] Philipp Haller and Martin Odersky. Event-based programming without inversion of control. In *Proc. Joint Modular Languages Conference*, Springer LNCS, 2006.

[Lip75] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[Lis88] Barbara Liskov. Distributed programming in argus. *Commun. ACM*, 31(3):300–312, 1988.

[LLS07] Yu David Liu, Xiaoqi Lu, and Scott F. Smith. Coqa: Concurrent objects with quantized atomicity (long version), October 2007. Available at `http://www.cs.jhu.edu/~xiaoqilu/Concurrency/`.

[Lom77] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *SIGOPS Oper. Syst. Rev.*, 11(2):128–137, 1977.

[LS89] Leslie Lamport and Fred B. Schneider. Pretending atomicity. Technical Report TR89-1005, Digital Equipment Corporation, 1989.

[Mil] Mark Miller. The E Language, `http://www.erights.org`.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *Compiler Construction: 12'th International Conference, CC 2003*, volume 2622, pages 138–152, NY, April 2003. Springer-Verlag.

[NMAT+07] Yang Ni, Vijay Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *ACM SIGPLAN 2007 Symposium on Principles and Practice of Parallel Programming*, March 2007.

[Sin89] Mukesh Singhal. Deadlock detection in distributed systems. *IEEE Computer*, 22(11):37–48, 1989.

[WHJ06] Adam Welc, Antony L. Hosking, and Suresh Jagannathan. Transparently reconciling transactions with locking for java synchronization. In *ECOOP'06*, pages 148–173, 2006.

[WJH04] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Transactional monitors for concurrent objects. In *ECOOP'04*, volume 3086 of *Lecture Notes in Computer Science*, pages 519–542. Springer-Verlag, 2004.

[YBS86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming abcl/1. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 258–268, New York, NY, USA, 1986. ACM Press.