COQA: A CONCURRENT PROGRAMMING MODEL WITH UBIQUITOUS ATOMICITY

by

Xiaoqi Lu

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

November, 2007

Abstract

This paper introduces a new language model, *Coqa*, for deeply embedding concurrent programming into objects. Every program written in Coqa has the desirable built-in behaviors of quantized atomicity, mutual exclusion, and race freedom. Such a design inverts the default mode of *e.g.* Java, where such properties have to be manually coded using primitive language constructs such as **synchronized**, resulting in programs that are vulnerable to surprising run time errors. A key property of Coqa 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. So rather than building atomicity locally, with small declared zones of atomicity, we build it globally, down from the top. We justify our approach from a theoretical basis by showing that a formal representation, KernelCoqa, has provable quantized atomicity properties. Then we extend KernelCoqa with two I/O models in which we conduct an in-depth study on how I/O affects the atomicity property and establish atomicity theorems with respect to I/O. We perform a series of benchmarks in CoqaJava, a Java-based prototype implementation

incorporating all of the KernelCoqa features, to demonstrate the practicability of the Coqa model. We give concrete CoqaJava examples of various common concurrent programming patterns which showcase the strength and ease of programming of the language.

Advisor: Professor Scott Smith, The Johns Hopkins University

Readers: Professor Robert Rynasiewicz, The Johns Hopkins University

Professor Yair Amir, The Johns Hopkins University

Acknowledgements

I would like to express the deepest appreciation to my advisor, Professor Scott Smith, for intellectual support and encouragement. Without his guidance and persistent help this dissertation would not have been possible.

I would like to thank my committee members, Professor Robert Rynasiewicz and Professor Yair Amir, who have been supportive to my research throughout the years.

I also wish to thank all members in the programming language group, especially Yu Liu for his invaluable inputs to my research and Paritosh Shroff for his constant encouragement.

I dedicate this dissertation to my parents, Zhiqing Lu and Shurong Dai, and my husband, Zhongtian Yan, for your endless support and love.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Today multi-core CPUs have become the general trend in computer hardware. This technology is transforming the personal computer into a powerful parallel machine and incents evolutions in every field of computer science, such as operating systems, libraries and applications. Programming languages are also involved in this architectural evolution because programming languages have to evolve in order to meet the programming demands of the times.

Software applications can benefit greatly from multi-core architectures if they are coded with multi-thread. However, the reality is that most applications are not written to use parallel threads intensively because developing multi-threaded applications is notoriously challenging. Due to the difficulties of thread synchronization and debugging threaded code, programmers are daunted to take advantage of the computation power of multi-core CPUs. We think that such a situation is caused by a gap between low level parallel hardware and high level programming languages. Therefore, designing programming languages

to match the coming ubiquity of concurrent computation becomes an imminent task.

The Coqa project investigates how a concurrent object-oriented language should be designed in the context of a pervasive concurrent computing environment. The ultimate goal of the Coqa project serves to reconsider the right concurrency model of a programming language, so that programmers can develop multi-threaded applications more easily. Consequently, more of the performance gained at the hardware level can be propagated to the application development level due to increasing usability.

1.1 Concurrent Programming and Correctness Properties

Concurrency is a property of a system in which there can be several tasks making processes at the same time. A task can be a heavyweight process or a lightweight thread. Concurrent systems differ from sequential ones in that tasks can potentially interact with each other [58].

Concurrency improves system responsiveness, maximizes resource usage and hence greatly increases the throughput of computers. A distinct challenge of concurrent programming is that programs are difficult to reason about due to the nature of non-determinism. There are an incredibly large number of interleaving possibilities across all possible runs of a program and how the program runs depends on the actual interleavings at run time. In this chapter, we review the desirable correctness properties.

1.1.1 Data Race Freedom

A *race condition* occurs when a shared field is accessed by two threads at the same time and at least one of the accesses is a write. Race conditions may lead to timing flaw in a system because the system behavior would depend on the order of the two accesses to the field at run time. Inconsistencies in current programs may stem from race conditions producing storage conflict at the level of the raw memory cell [42] and cause serious damage to a system.

```
public class Even {
    private int n = 0;
    public int next() {
        ++n;
        ++n;
        return n;
}
```

Figure 1.1: Race Condition

Thread A	Thread B
read 0	
write 1	
	read 1
	write 2
read 2	read 2
	write 3
write 3	return 3
return 3	

Table 1.1: Race Condition Trace

The example in Figure 1.1 from [42] shows how a race condition can lead to dan-

gerous data inconsistency. The class `Even` is supposed to return an even number whenever its `next` method is called. However, if multiple threads race on mutating the shared variable `n`, then the output can even be a value that was not ever written by any thread. For instance, consider the following possible execution trace shown in Table 1.1, in which reads correspond to `getfield`s of Java bytecode and writes correspond to `putfield`s. At the end of the invocation on the method `next`, both threads `A` and `B` get 3,a non-even number.

A decade ago, race conditions were a less severe problem because most systems were sequential programs. But today's computer hardware has greater concurrency capability via multi-core technology. Accordingly, multi-threaded applications are becoming the future of software. Any resource sharing among threads creates possibilities for race conditions. With the ever-growing application sizes, data race is indeed a common vulnerability in software and it can potentially lead to catastrophic disasters in real world systems. For instance, the Northeast Blackout of 2003 affected an estimated 50 million people in United States and Canada and caused around $6 billion in financial losses. This outage was later found to be caused by a race condition bug in the computer system. This incident warns us of the harsh reality and challenge we are facing today.

Race conditions can be eliminated by using locks. A thread acquires a lock before accessing a shared resource and releases the lock when it is done with the resource. Any multi-threaded programming language must provide language primitives to avoid data races. Java guarantees methods and statement blocks declared with the **synchronized** keyword are accessed exclusively by a single thread at any given time. Theoretically, every Java object is implicitly associated with a lock. Before entering a synchronized method a thread

needs to first acquire the lock on the current **this** object, while a synchronized block can specify which object's lock should be acquired before the block can be run. For instance, adding a **synchronized** method modifier to the `next` method in Figure 1.1 successfully removes the race condition in this particular example.

Not all data races are malicious. There are cases where strict data consistency is not required and some systems may introduce or leave data races untreated for the purpose of responsiveness and performance. For instance, a web server may allow users to request pages undergoing modifications if the completeness of these pages can be tolerated by users. We are not concerned about these benign data races, but rather those causing unwanted system behaviors. *Data race freedom* is a property where race conditions do not arise. In a race free system, threads compete for shared resources and then use them in a mutually exclusive way so that when a thread completes its operations on a resource, it leaves it in a consistent state for other threads.

1.1.2 Mutual Exclusion

Mutual exclusion is a means of making sure multiple threads take turns to access shared resources. Code executed in such a mutually exclusive way is called a *critical section*. As mentioned before, mutual exclusion is commonly used to eliminate race conditions. However, mutual exclusion can be a property not tied to any specific fields, but a property of arbitrary code blocks. To avoid data races on a field, *all* accesses to the field should be mutually exclusive.

With the **synchronized** keyword, Java programmers can code their applications to exclude race conditions, but they have to follow very strict programming discipline to

make it right. Figure 1.2 shows an example given in the Sun Java thread tutorial [12]. In this example, a producer thread and a customer thread write and read a `CubbyHole` object respectively.

```
public class CubbyHole {
    private int content;
    private boolean available = false;

    public synchronized int get() {
        while(available == false){
            try {
                wait();
            }catch(InterruptedException e){}
        }
        available = false;
        notifyAll();
        return content;
    }

    public synchronized void put(int value){
        while(available == true) {
            try{
                wait();
            }catch(InterruptedException e) {}
        }
        content = value;
        available = true;
        notifyAll();
    }
}
```

Figure 1.2: CubbyHole

The `get` and `put` methods in Figure 1.2 are **synchronized** and hence at most one thread can be executing the two methods of a `CubbyHole` object. Specifically, when the first thread reaches the program point of method `get`, it acquires a lock on the **this** object

and the lock would be released when the thread exits the method. While the lock on **this** object is held by a thread, any other threads requesting the same lock would fail to acquire it. The `wait` and `notifyall` methods are defined by the Java root `Object` class. They are used to put a thread into a wait state and to wake up threads that are waiting on an object respectively. So the code in Figure 1.2 specifies if a thread cannot acquire the lock on a `CubbyHole` object, it will wait for the lock to be freed by its owner and then try to acquire the lock again, until it eventually gets a hold of the lock.

Note that the field `content` is **private** and it is accessed only via the `get` and `put` methods. Therefore, declaring the `get` and `put` to be **synchronized** is adequate to make sure that there is no data race on the field `content`. If `content` is a **public** or **protected** field, then there is no such guarantee because threads can access this field without using the properly synchronized `get` and `put` methods. Similarly, if the `CubbyHole` class has any unsynchronized method that also accesses the private `content` field, the same data race problem would arise. The root of this problem is that the Java **synchronized** keyword only ensures a semantics of mutual exclusion. It is the programmers' full responsibility to ensure that a shared field is always accessed in a mutually exclusive way.

1.1.3 Atomicity

Data race freedom is important to multi-threaded applications. However, it is neither a necessary nor a sufficient condition of program correctness [24]. Two examples, Figure 1.3 and Figure 1.4, illustrate this assertion.

In Figure 1.3, `balance` is the only object field declared in the `Bank` class and there is obviously no race on accessing it. Suppose the initial value for `balance` is 10 and there

```
class Bank {
    private int balance;
    void synchronized deposit(int n) {
         balance = balance + n;
    }
    int synchronized read(){
        int r;
        r = balance;
        return r;
    }
    void withdraw(int n) {
        int r = read();
        synchronized(this) {
            balance = r - n;
        }
    }
}
```

Figure 1.3: Race Free Is Not Sufficient

are two threads: one calls method `deposit(10)` and the other calls `withdraw(10)`. The expected final value of `balance` is 10. However, it is possible that the final value of `balance` turns out to be 0 if the withdraw thread reads the initial value of `balance`, 10, and stores the value into the local variable `r`. Then the deposit thread grabs the lock on the bank object and increases the value of `balance` to 20 and releases the lock. Then the withdraw thread acquires the same lock and resumes its computation of `balance` based on staled value stored in `r`, giving an incorrect result of 0.

On the other hand, Figure 1.4 shows accesses on the field `i` causing a race condition. One thread holds the `lock` object reading and writing `i`, while another thread reads `i` via the method `read`. However, the program behaves correctly in a multi-threaded context. The value of `i` always appears to increase by 1, atomically. This example demonstrates that race freedom is not necessary for program correctness.

```
class Ref {
    Lock lock = new Lock();
    private int i;

    void inc() {
        int t;
        synchronized (lock)  {
            t = i;
            i = t + 1;
        }
    }

    int read() {
        return i;
    }
}
```

Figure 1.4: Race Free Is Not Necessary

These examples clearly demonstrate that data race freedom is not enough to guarantee program correctness, a stronger property is needed. The key is *atomicity* – i.e., the property that a block of code can always be viewed as being executed sequentially no matter what interleaving takes place. The common technique to prove the atomicity property of a code block involves showing that for all execution paths in presence of concurrent executions, we can find an equivalent serial execution with the same behaviors. We will use the same technique in proving our theorems in later sections.

1.2 Atomicity in Existing Programming Models

Most existing programming languages provide language primitives/libraries for programming concurrent applications. But the more interesting question here is if they have adequate support for atomicity. Here we study this problem in three different language

models: Java as an example of an object-oriented model, the Actor model, and Software Transactional Memory systems.

1.2.1 Java Language

Existing object-oriented programming languages fall short of supporting concurrent programming because they are primarily designed for sequential programming. For instance, the focus of Java was originally portable Internet applications. There is no built-in language support for atomicity in Java. It is the programmers' sole responsibility to protect shared data, eliminate harmful interleavings, and build atomicity in their applications manually.

Figure 1.5 illustrates some subtleties of achieving atomicity in Java. The **synchronized** declaration of the method `foo` seemingly always gives us a serial execution. However, this method is not atomic. In a concurrent context, the two calls to `o.read()` can yield different results. The lower part of Figure 1.5 illustrates an interleaving of two threads which do not have an equivalent serial execution path.

The reality is that it is very difficult to code and debug multi-threaded applications using programming languages like Java because these languages do not have built-in support for atomicity. Programmers have to deal with exponentially many interleavings of threads. For instance, if the field `content` of the `CubbyHole` class is **public** or **protected**, it is difficult to locally reason about atomicity properties of code because all classes that are able to access this field directly or indirectly need to be considered. So getting atomicity correctly encoded in their applications requires programmers to have an in-depth understanding about subtle atomicity semantics, a topic we will discuss in Section 1.2.4.

```
class A {
    void synchronized foo(B ob) {
        ob.read();
        ob.read();
    }
}
class B {
    int f = 0;
    ...
    synchronized int read() {
        return f;
    }
    synchronized int write(int v) {
        f = v;
    }
}
```

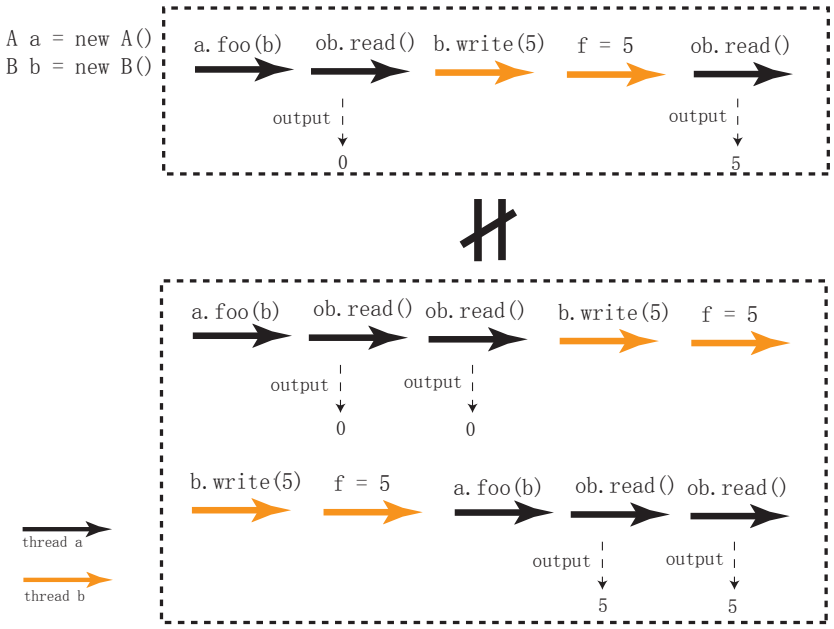

Figure 1.5: Atomicity in Java

1.2.2 The Actor Model

The *Actor* model [1] is a mathematical model of concurrent computation for open distributed systems. Actors are autonomous components that communicate with each other asynchronously. Each actor has its own private store. An actor can create another new actor and send and receive messages. All these operations are asynchronous which means an actor does not block on any sends waiting for replies. The Actor model makes sense for distributed programming in which communication is by nature asynchronous.

Actors [1, 2] have atomicity deeply rooted in their model. Due to the fact that communication among actors is exclusively by asynchronous messaging, atomicity is inherently preserved for every method. Once the execution of a method is initiated, it does not depend on the state of other actors, and each method is therefore trivially viewed as executed in one step no matter what interleavings occurred. The Actor model's per-method atomicity is a local property in the sense that it neither includes more than one actor nor other methods invoked by the current method.

Actors are a weak programming model for tightly coupled concurrent code where synchronized communication is frequently needed. Because the Actor model does not have a built-in synchronized messaging mechanism, programmers need to code synchronized communication manually. A simple task such as an actor sending a message to another actor and waiting for a reply to resume its own task becomes a heavy coding job. The common technique for accomplishing synchronous interaction in an asynchronous model is to use continuations: the computation scheduled for being executed after receiving a reply would be wrapped up as a continuation, and then the sender actor waits exclusively for the reply

from the receiver to trigger the execution of the continuation before it can handle any other received messages.

1.2.3 Software Transactional Memory

Software Transactional Memory (STM) is a direct application of the commit-rollback mechanism in database systems to programming languages. Basically, in transactional programming languages, programmers can label code blocks to be "atomic". Then it is the language runtime's responsibility to ensure these labeled code blocks satisfy certain atomicity semantics by using the rollback mechanism. If data access violations are detected before a transaction can commit the effects of its computation, the transaction will be abandoned and the system rolls back to the initial state before the abandoned transactions start. An access violation happens if one transaction writes to a memory location while another transaction reads from the same location. A *transaction* here is one run time execution path of a code block labeled as atomic. STM has been adopted by many systems such as [31, 64, 32, 57, 13, 3, 19]. Because STM systems have the language support for atomicity, they are a better model for concurrent programming than languages like Java that lack of any form of atomicity.

In order to make sure that the system can roll back to the state before an abandoned transaction, a transactional system needs to do bookkeeping of the initial state of every transaction. The overhead of such bookkeeping can grow unexpectedly large with an increasing number of threads and transaction sizes. So STM programmers are encouraged to limit the size of atomic blocks to avoid large run time overhead.

1.2.4 Atomicity Semantics

The seemingly straight-forward definition of atomicity conceals different semantics with subtle differences in various language models. Understanding what atomicity really means in a system is crucial for writing correct applications.

Ubiquitous Atomicity or Selective Atomicity Different forms of atomicity exist in the Actor model and STM systems. The Actor model has *ubiquitous atomicity*: every method in the Actor model is atomic and it is not possible to code a method in a non-atomic way. On the other hand, STM systems have *selective atomicity* in the sense atomicity is only guaranteed if a code block is *explicitly* labeled as atomic. Otherwise, no concern for data access violation ever arises.

Weak Atomicity and Strong Atomicity In a transactional programming language, an application may include both transactional and non-transactional code. If the system only considers atomicity among transactions so that it is possible for non-transactional code interleaving with transactions, then the system has a so-called *weak atomicity. Strong atomicity* on the other hand guarantees that atomicity is respected among transactions as well among transactions and non-transactions. A study of weak and strong atomicity in a transactional language model can be found in [10].

Weak atomicity, the atomicity semantics supported in most STMs, can be potentially dangerous because it is hard to reason about program correctness if a transaction is atomic only in respect to other transactions but not other non-transaction executions. It only gives programmers an illusion of atomicity which could trap them into a hard-to-detect

bug. The subtlety of weak and strong semantics hinders programmers' understanding of STM systems and weakens their capabilities of using these systems correctly.

The Actor model has strong atomicity in the sense that there is no execution step falling outside an execution of a method while every method is an atomic block.

Shallow Atomicity and Deep Atomicity Although the atomicity supported in the Actor model is ubiquitous and strong, it is also shallow. Each message is the unit of atomicity and it only consists of local computation of one actor. Specifically, for a task involving multiple actors and spanning multiple messages, the best atomicity we can get in the Actor model is atomicity in the unit of each message. It is not possible to obtain a bigger zone of atomicity by relying on the language model itself. Such *shallow atomicity* is limited in terms of programmability because the demand of atomicity over multiple messages and objects is not uncommon in object-oriented programming.

Atomicity in many STM systems on the other hand is *deep atomicity*. It can span the run time execution of an atomic block which may across object and method boundaries. Those systems usually dynamically keep track of memory locations that have been read/written by a transaction till a commit/abort checkpoint is reached.

1.3 Coqa: A New OO Language for Concurrency

In this thesis, we propose a new object-oriented language for concurrent programming that provides strong language support of a desirable atomicity property for all programs. We name this new object-oriented language, Coqa (for *C*oncurrent *o*bjects with *q*uantized *a*tomicity). We here discuss the rationales of the important design choices we have

made for Coqa. Coqa is an object-oriented programming language with a so-called *atomicity by design*. Atomicity is built into the language syntax and semantics by proactively limiting sharing. Coqa also has built-in support for mutual exclusion and race freedom. With these concurrency properties, Coqa eliminates the need for a complex memory model like what Java has [48], a cumbersome memory model with a lot of intricate complications.

1.3.1 Quantized Atomicity: Ubiquitous, Strong and Deep

One of Coqa's main goals is to significantly reduce the number of interleavings possible in concurrent program runs and to significantly ease the debugging burden for programmers. If two pieces of code each have 100 steps of execution, reasoning tools would have to consider interleaving scenarios of 2^{200} possibilities. However, if the aforementioned 100 steps can be split into 3 atomic zones, there are only 20 possibilities. We call these atomic zones *quanta* in Coqa. Steps of each quantum are serializable regardless of the interleaving of the actual execution.

The concurrency unit in our language is a *task*. This unit of execution can potentially interleave 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. A key property of Coqa programs is that object methods will often have strong atomicity over the whole method by design. A task spanning those methods is then atomic by default. But, for tasks that simply cannot be atomic, our model allows programmers to relax the full atomicity by dividing a task into a small number of quanta, giving a model with *quantized atomicity*. Because every quantum is delimited by programming points as explicitly specified by programmers, programmers only need to consider

how to logically partition a program in one or more zones. Coqa guarantees the atomicity property of those zones.

Quantized atomicity of Coqa is ubiquitous. The default mode of Coqa is that the overall atomicity is *preserved* for every task unless programmers *explicitly declare otherwise*, which demarcates a task into a sequence of quanta and every quantum is atomic. Quantized atomicity of Coqa is strong because all code has to be in one atomic zone or another. Quantized atomicity of Coqa is deep. An application can be divided into multiple quanta, each of which spans execution steps that may involve many objects and across multiple methods. So quanta can be much bigger atomic zones than the per-actor-per-message shallow atomic zones inherent to the Actor model.

1.3.2 Atomicity by Locking vs by Rollbacks

Quantized atomicity gives a programming language model a notion of ubiquitous atomicity which has very strong reasoning power. In terms of how atomicity can be achieved in a language model, there are in general two approaches: the *pessimistic blocking approach* and the *optimistic rollback approach*. Coqa takes the first route.

The rollback/commit scheme was first proposed by [40] to increase concurrency while at the same time preserve the atomicity property of transactions in database systems. It addresses throughput and deadlock problems in databases. Databases are fundamentally open systems that need to deal with all possible inputs. A database system cannot predict with certainty the pattern of how it will be accessed because any application with an interface to the database can access it and the application could change its access pattern any time without any notice. So it is impossible to design a general purpose deadlock

prevention technique for database systems. On the other hand, software applications are closed systems. The complete code of an application is usually available before it can be deployed. Therefore, analyzing applications statically for deadlocks is a more realistic and efficient approach in a programming language system. Coqa is likely to have more dealocks due the increased amount of mutual exclusion needed for quantized atomicity. It significantly raises the importance of developing quality static deadlock prevention tools. There has been extensive research on detecting deadlocks statically as we will discuss as future work in Section 6.

STM systems directly adopt the rollback/commit mechanism in database systems into programming languages. It is an approach that applies a solution more suitable to an open system to a rather closed system. We believe that this approach only offers a partial solution to achieve atomicity in a programming language because STM systems have not acknowledged the fundamental differences between a database system and a programming language. The rollback approach has known difficulties on rolling back certain forms of computations, I/O in particular. Therefore, the rollback approach is not a general strategy for achieving ubiquitous atomicity even thought it can be efficiently applied to a small amount of code blocks in a large application. For instance, the need to roll back a network message sent across political boundaries opens up a Pandora's Box of problems. This fatal shortcoming directly clashes with our proposed quantized atomicity because it is not possible to have a ubiquitous notion of atomicity in the presence of I/O, if the optimistic approach is taken. Due to the inherent mutual exclusion between I/O and the rollback approach, STM systems unavoidably have to exclude I/O programming in atomic blocks.

However, as an implementation technique, the rollback approach can be used in Coqa as an underlying implementation strategy for regions where there is no I/O, just not everywhere. Section 6 discusses a hybrid approach in more detail.

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 [13] and `onAbort` *etc.* methods in [53]) as user-definable handlers. These handlers themselves may introduce races or deadlocks, so some of the appeal of the approach is then lost [53]. More importantly, such design blurs the language semantics for atomicity. It makes reasoning about the correctness of transactional blocks more difficult.

Transaction-based systems do not have deadlock problems but they have to deal with *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. Livelocks are less server than deadlocks in the sense that the possibility of eventually resolving a livelock can be statistically high with multiple retries so that a sytesm can eventually recover from a livelock and progress, while a deadlock would cause the system forever.

1.3.3 Integrating Concurrency with Object Interaction

Existing object-oriented languages like Java use a non-object-based syntax and semantics for concurrent programming. 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 [47].

Coqa however integrates concurrency and object interaction seamlessly. More specifically, in addition to the familiar `o . m(v)` message send expression, Coqa provides two more messaging primitives: `o -> m(v)` for introducing concurrency into a system and `o => m(v)` for explicitly introducing interleavings between concurrent executions for the purpose of communication. Our goal is to integrate concurrency naturally into an object-oriented language model that at the same time delivers a powerful atomicity semantics to programmers so that concurrent applications can be easy to write and easy to understand.

1.4 This Dissertation

In this dissertation, we present a new programming language model for concurrent programming called Coqa. Chapter 2 introduces how Coqa takes advantage of basic object-oriented interactions to express important concurrent programming primitives. In particular, different forms of message passing are all that are needed to express creating a separate task, intra-task messaging, and inter-task communication. In this chapter, we also informally present the implications of different types of message send in terms of atomicity. We formalize the Coqa model in a core language called KernelCoqa in Chapter 3, and prove the model has the properties of quantized atomicity, mutual exclusion, and race freedom. Chapter 4 extends KernelCoqa to a more realistic language model with the capability of modeling real world I/O. We propose two different I/O models, and rigorously formalize that desirable concurrency properties still hold with I/O. Chapter 5 describes 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 re-

mains unchanged. Preliminary benchmarks show that CoqaJava has reasonable overhead with only basic optimizations applied, which indicates a good performance potential in a more complete implementation. Chapter 6 studies possible approaches that can further improve the Coqa model in various aspects. Chapter 7 gives concrete CoqaJava code examples of various concurrency patterns to demonstrate the strength and ease of programming of the language.

Chapter 2

Informal Overview

In this chapter, we informally introduce the key features of Coqa in an incremental fashion, using a simple example of basic banking operations, including account opening and balance transfer operations. Figure 2.1 and Figure 2.2 give the barebones version of the source code we start with. Bank accounts are stored in a hash table, implemented in a 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 Figure 2.3. For brevity here we present simplified code which omits checks for duplicate keys, nonexistant accounts, and overdrafts.

2.1 The Barebones Model

First we describe a barebones model of Coqa to illustrate basic ideas such as task creation and object capture.

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

        class Bank {
            public void transfer (String from, String to, int bal) {
                Status status = new Status();
(A1)            status.init();
(A2)            Account afrom = (Account)htable.get(from);
(A3)            afrom.withdraw(bal, status);
(A4)            Account ato = (Account)htable.get(to);
(A5)            ato.deposit(bal, status);
            }
            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 2.1: A Banking Program: Version 1

```
        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 2.2: HashTable

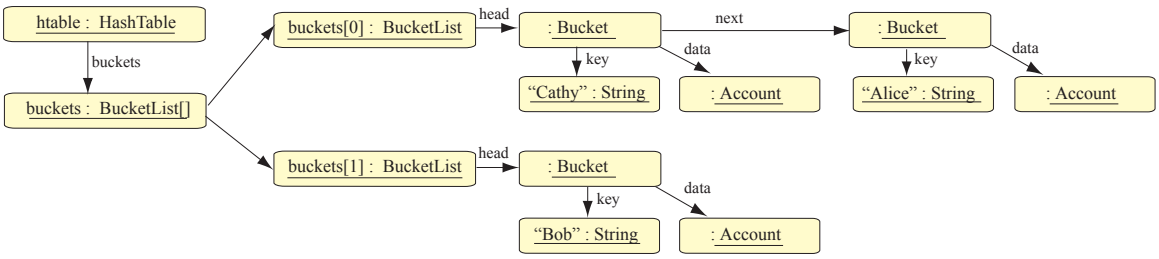

Figure 2.3: HashTable: a UML Illustration

2.1.1 Task as Concurrency Unit

In Coqa, we model concurrent computation as a number of threads (in our terminology, *tasks*) competing to "capture" objects, obtaining exclusive rights to them. 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 using `->` syntax. For example, in Figure 2.1, 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 the object `bank` with the indicated arguments. Asynchronous message send returns immediately, so the sender can continue, and a new task then is created to execute the invoked method. This new task terminates when its method is finished.

Tasks are selfish in Coqa. Figuratively, Java threads selflessly *share* objects in

the heap by default. Unfortunately such an idealistic model breaks down in the face of concurrency: mutual exclusion, race freedom and atomicity, which all require zones of exclusive use of objects, and thus threads need to be orchestrated carefully to be selfish in their control of objects. In our language, tasks always selfishly *compete* for objects in the heap and such a selfish mechanism better satisfies selfish properties like atomicity.

The basic idea for Coqa tasks to be selfish is simple. Whenever a task accesses an object which has not yet been captured by other tasks, 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*). There are different modes that a task may capture an object. But for now, let us assume that when an object is captured by a task, it becomes exclusively accessible only to this task. We will discuss more details about different types of capture in the coming sections. Objects in the capture set of a task remain *captured* until they are all freed when the task ends. If a task intends to access an object that has already been captured by some other task, it is blocked at that point of execution until the needed object is freed and it successfully captures the object.

Capturing is a blocking mechanism, but unlike Java where programmers need to explicitly specify what to lock, where to lock and when to lock, the capture and blocking of objects is fundamentally built into Coqa. It is carried out by Coqa automatically and implicitly for all tasks.

2.1.2 Timing and Scope of Capture

The description of how Coqa tasks capture objects during their execution is not complete before we specify *when* an object is captured and *how selfish* captures are.

In regard to timing, we could decide to capture an object when a task sends a message to it, or when the object's fields are accessed by a task. The first candidate is a more intuitive choice in that it allows programmers to reason in an object-based fashion rather than a per-field fashion. The second candidate is more an optimization strategy to improve performance since it potentially allows more parallelism inside an object. For instance, consider `transfer` method of Figure 2.1. When the programmer writes line `A2`, his/her intention is to selfishly capture the `HashTable` object referenced by `htable`. Synchronous messaging at a high level expresses the fact that the task will be accessing the receiver. However, this view is more conservative than is necessary. 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. In the aforementioned example, capture of the `htable` by the invoking task occurs in the middle of executing the `get` method of the `HashTable`, when the field `buckets` is accessed. It is for the same reason that we can have parallelism in `M1` and `M2`, since the moment of the messaging does not capture the `bank` object. Running both concurrently will not lead to competition for it since the `transfer` method does not access the `bank` field at the point when the two messages are delivered to the `bank` object.

The second question is related to field access itself. The selfish model we have thus far described treats read and write accesses 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. If the `transfer` method also included some other time-consuming operations after it reads data from the `htable` object, different tasks of `transfer` would all block on the `HashTable` object and would make it a bottleneck for concurrency. In fact, concurrent read

accesses alone are perfectly fine, things only become problematic upon field writes. In most cases, read operations are more frequent than write operations. As a further refinement, our notion of capture is further developed to use the standard two-phase non-exclusive read lock and exclusive write lock [28] approach. When an object's field is read by a task, the object is said to be *read captured* by the task; 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. When write captured, the object has to be exclusively owned, *i.e.* not read captured or write captured by another task.

One possible execution sequence of the task created by line `M1` of Figure 2.1 is illustrated in the first column of Figure 2.4. As the execution proceeds, more and more accessed objects are captured via field reads (yellow or light grey boxes) or writes (red or dark gray 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 may also cause deadlocks. 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 `bal += b` expression is indeed two operations: read from `bal`, then write to `bal`). The `Account` object of `Alice` would have read locks from both `M1` and `M2`. Neither party can add a write lock to it. 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

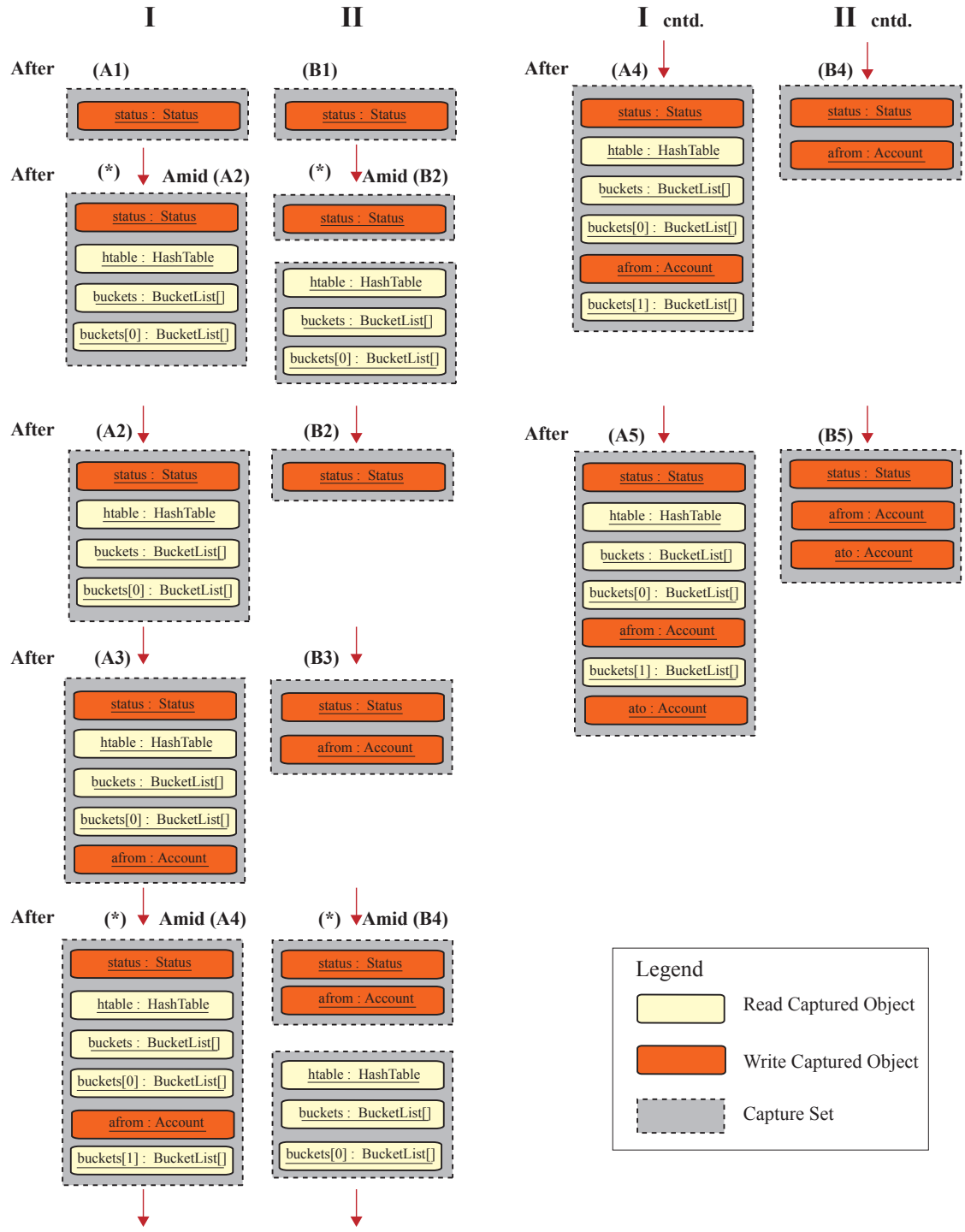

Figure 2.4: Execution Sequence for the Task Created by Figure 1 Line `M1`

declared as **exclusive class** `Account` {...} and that would have prevented deadlock from arising.

Performance of the two-phrase locking is sensitive to application implementations. It works more efficiently with certain access patterns such as data objects that are infrequently modified. Because Coqa is a model with more pervasive locking, it is important for Coqa to have an implementation with a performance matching with other object-oriented programming languages. In practice, we believe the two-phase locking is a better scheme for Coqa because there are more field reads than writes in most applications. Since extra syntax is provided to declare objects to be always captured in an exclusive way as discussed above, Coqa can be very flexible for meeting different programming needs for various data accessing patterns.

2.1.3 Concurrency Properties

The barebones model has good concurrency properties for tasks running in parallel: quantized atomicity, mutual exclusion, and race freedom. In this section, we discuss how these properties are held for tasks informally. Formal proof is given in Section 3.3.

A task creates a new task by a `->` message. In the example of Figure 2.1, the default bootstrapping task t_{main} starts from the special `main` method and it creates two child tasks t_1 and t_2 in lines `M1` and `M2`. The conflict of interest between t_1 and t_2 is on accessing Alice's `Account` object. If t_1 arrives at the access point and captures the object first, it would have the exclusive access ownership on this object until it finishes. The later arriving t_2 would have to block until the t_1 is done. So, it is easy to see that t_1 and t_2 can always be viewed as one executing before the other.

This example can be easily extended to more general cases. For two tasks t_1 and t_2 none of which create other tasks, intuitively, if they do not read/write any shared object at all, they can be arbitrarily interleaved without affecting each other. If they do access some shared objects, they then need to compete for them. At any particular time and no matter how t_1 and t_2 have interleaved up to the moment for first object contention, we know the objects t_1 have accessed must either be read captured or write captured by t_1. It is obvious that t_2 must not have write captured any of these objects due to the fact that t_1 is able to capture the objects. In other words, all of these objects are not mutated by t_2. This demonstrates that tasks t_1 and t_2 has several appealing consequences no matter how they interleave.

Atomicity : Since t_2 has no (mutation) effect on the objects t_1 cares about, all computations performed by t_2 can be equivalently viewed as non-existent from t_1's perspective. Task t_1 can then be viewed as running alone. The same can be said about t_2. Together this implies that the execution of t_1 and t_2 can be serialized. In this case, t_1 and t_2 are both atomic: each of them has only one quantum, a special case of quantized atomicity.

Mutual Exclusion : Following the same reasoning, t_1 and t_2 have no (mutation) effect on each other's captured objects.

Race Freedom : The mutual exclusion of t_1 and t_2 guarantees that there is no race to access any object field.

2.1.4 Task Creation as Quantum Demarcation

t_1 and t_2 in the discussion of the previous section do not create other tasks. This assumption is needed because an asynchronous message send is a *demarcation point* where the current quantum ends and a new one starts.

```
        public class Main {
            public static void main (String [] args)  {
                Foo foo= new foo();
                ...
(L1)            foo -> set(300);
                ...
                foo.set(0);
                ...
          }
        }

        public class Foo {
            private int fd;
            public void set(int v)  {
                ...
(L2)            fd = v;
                ...
            }
        }
```

Figure 2.5: Task Creation Demarcates Quanta

Consider the example in Figure 2.5 where there are two tasks at runtime: the bootstrapping main task t_{main} and t, a child task created by t_{main} in line `L1`. Notice that the two tasks have contention in line `L2` since both of them try to write access the same object `foo`. If the child task t reaches line `L2` before t_{main}, t_{main} would have to wait until t finishes. This would put t_{main} observably after t. However, the object `foo` is created by t_{main}, in order for t to access this object, t_{main} has to be run before t. This essentially means that we cannot serialize t_{main} and t so that one seemingly happens before the other,

each as a whole quantum. The reason is that t_{main} no longer has just one atomic quantum, but rather two quanta splitting at the point where t is created (line `L1`). Namely, t_{main} has the property of quantized atomicity with two quanta.

It might seem harmless to view that a child task runs before its parent if they do not access any shared object. However, such a violation of task creation makes a language model unimplementable because a computer system would have to realize what will happen in the future before what is happening now.

Now, we can extend the atomicity property of Coqa to a more general case:

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

The fact that task creation demarcates a quantum of the parent task does not significantly reduce the power of the Coqa model. In most concurrent programming patterns, there are one or a few threads whose main jobs are to create other working threads for performing real computations. So in practice, the task creation hierarchy is rather shallow and most tasks we care about are peer tasks. For instance, the t_{main} running the `main` method in Figure 2.1 is split into multiple quanta at program points where new tasks are spawned. However, t_{main} simply creates tasks for doing the real work. How many quanta t_{main} has is not the focus of the application.

2.1.5 Better Threading Syntax

Task creation via `->` is a more natural and flexible syntax than Java's. There are two ways to create a new thread in Java. One approach is to define a subclass of the

`java.lang.Thread` class, override its `run` method, and then create an instance of it. The other way is to instantiate a `java.lang.Thread` object with a target object whose class implements the `java.lang.Runnable` interface in which a single `run` method is defined. After a `Thread` object is created, the `start` method is invoked so that the Java Virtual Machine can schedule the thread. Obviously, Java threading requires some explicit coding, while the only code needed in Coqa is a `->` message send. No extra wrapper class or method declaration is needed.

Moreover, the `run` method that a Java thread starts from takes no arguments, which means a Java thread has to get its initialization data in some other ways indirectly, for instance, by accessing some global fields. In general, such indirect data flow is not a good programming practice because it encourages data sharing instead of data encapsulation. Such problem does not exist in Coqa because asynchronous messages can carry necessary initial values to a new task and types of these arguments are explicitly declared as method declarations, a much typed way to start a new thread of execution.

2.2 Subtasking: Open Nesting

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` in Figure 2.2. Let us suppose when the account of `Dan` is added, it will become the head `Bucket` of `bucket[0]` in Figure 2.3. 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 on the object of `bucket[0]`. `M3`, the task of adding `Dan` as a new account, can only be achieved after the completion of `M1`, although it is a task totally unrelated to the 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 life cycle 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 `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.

2.2.1 Subtasking for Intentional Interleavings

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 Figure 2.4 to the following Figure 2.6 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();
(B1)        status.init();
(B2)        Account afrom = (Account)htable ⇒ get(from);
(B3)        afrom.withdraw(bal, status);
(B4)        Account ato =(Account)htable ⇒ get(to);
(B5)        ato.deposit(bal, status);
        }
```

Figure 2.6: `Bank`'s `transfer` Method: Version 2

Subtasking is still a synchronous invocation. 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.

The second column of Figure 2.4 illustrates a possible execution sequence for the same task `M1` when the `transfer` method is changed to the one in Figure 2.6. When `htable ⇒ get(from)` is invoked, a subtask is created. Internal to the subtask execution, the subtask still captures objects as a task would do. For instance, in the middle of the `get` method during bucket lookup, two separate capture sets are held. The `transfer` task keeps its own object `Status` object, and all `HashTable`-related objects are put in the capture set of the subtask. When the method invocation `htable ⇒ get(from)` completes, the task of opening an account for `Dan` can now access the `HashTable`, rather than being required to wait until `M1` has completely finished.

A subtask is also a task in the sense that it prevents arbitrary interleaving and quantized atomicity is preserved for subtasks. The change in line `B2` admits some inter-

leaving 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 related to open nesting in transaction-based systems [53, 13]. Open nesting is used to nest a transaction inside another transaction, where the nested transaction can commit before the enclosing transaction runs to completion. (Nested transactions that cannot commit before their enclosing transactions are commonly referred to as closed nesting.) Open nesting of transactions can be viewed as early commit, while subtasking can be viewed as early release.

2.2.2 Capture Set Inheritance

One contentious issue for open nesting is the case where a nested transaction and the transaction enclosing it both need the same object. In Atomos [13], this 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 and add the restriction that a subtask never accesses an object held by its enclosing tasks. This however would significantly reduce programmability. First of all, programmers have to keep track of object accesses among tasks and subtasks. And they have to be concerned about data sharing together with the relationship between tasks all the time, which we think overloads programmers with an unnecessary burden. Moreover, disallowing overlap of capture sets between a task and its

subtask potentially introduces more possibility of deadlock. 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 account access history. The `Status` object is captured by the `transfer` task via invoking the `init` method. Suppose the `HashTable` class also accesses the `Status` objects in its `get` method, for example recording the retrieval time of an account object. Had we disallowed the access of `Status` from the `HashTable` task, the program would deadlock.

In Coqa, subtasks' accessing their creating tasks' captured objects 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 synchronous. So there is no concern for 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.

2.2.3 Quantized Atomicity with Subtasking

The presence of subtasking in a method delimits its atomic quanta. The objects captured by the subtask in fact can serve as a communication point between different tasks which will split atomic zones of the two. For a more concrete example, consider two tasks t_1 and t_2 running in parallel. Suppose task t_1 creates a subtask, say t_3, and in the middle of task t_3, some object `o` is read captured by the subtask t_3. According to the definition of subtasking, `o` will be freed at the end of t_3. t_2 can subsequently write capture it. After t_2 ends, suppose t_1 read captures `o`. Observe that object `o`'s state has been changed since

its subtask t_3 had previously read o. Task t_1 thus cannot assume it is running in isolation, and so it is not an atomic unit.

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

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

The fact that subtask creation demarcates one atomic quantum into two smaller ones weakens the quantized atomicity property of the barebones system. But as long as subtasks are created only when they are really needed, the atomic quanta will still be considerably large zones of atomicity. This is a significant improvement in terms of reducing interleavings. More specifically, interleavings in Coqa occur in units of quanta, while languages like Java interleavings by default happen between every single statement. In reality, what really 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. Moreover, mutual exclusion and data race also hold as in Section 2.1.

messaging	why is it	why should you use
`o . m(v)`	intra-task messaging	promotes mutual exclusion and atomicity
`o -> m(v)`	task creation	introduces concurrency by starting a new task
`o => m(v)`	subtasking	promotes parallelism by encouraging early release

Figure 2.7: The Three Messaging Mechanisms and Their Purposes

2.2.4 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 for, in Figure 2.7. 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` methods in Figure 2.1 and in Figure 2.6 are valid programs, depending on what the programmer believes is needed for the `transfer` method, full atomicity with one quantum or quantized atomicity with multiple quanta. 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 in Figure 2.8.

```
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 2.8: `Bank`'s `transfer` Method: Version 3

This version of `transfer` has more concurrency, as a result of a more relaxed data consistency need, between tasks created in lines `M1` and `M2` in Figure 2.1. Right after the task `M1` withdraws from `Alice`'s account, the account object becomes available for the task `M2` to deposit rather than being blocked until the task `M1` completely finishes. In this case, suppose there is an audit task counting the total balance of all bank accounts. The task reads Bob's balance and then Alice's account after it is freed by the task `M1`. Then the audit task might get an incorrect total balance because the balance transfer carried by the task `M1` is only half-way through: some money has been withdrawn from Alice's account but not yet deposited to Bob's.

2.3 I/O Atomicity

It is crucial for a programming language to incorporate a proper I/O model that reflects a realistic way of interaction with the outside world. We introduce two different ways to model I/O in Coqa: the fair I/O model and the reserveable I/O model. In both models, behaviors of I/O are captured by special I/O objects. This is similar to object-oriented languages such as Java where, for instance, `System.in` is an I/O object representing standard input.

I/O objects in the two models differ in the types of message they can receive. In the fair model, only `->` and `=>` messages can be sent to an I/O object, while in the reserveable model, an I/O object can handle all types of messages including local synchronous (.) messages. The reserveable I/O model subsumes the fair I/O model. If all I/O objects in the reserveable model are only sent `->` and `=>` messages but never local synchronous (.)

messages, then the reserveable model is essentially equivalent to the fair model. However, we believe it is important to study and compare the two models head-to-head because the fair I/O model is the common I/O abstraction adopted by existing object-oriented languages such as Java. Yet the reserveable I/O model goes one step further to achieve stronger atomicity properties in terms of I/O operations.

2.3.1 Fair I/O

I/O objects are built-in system objects in the fair I/O model. They encapsulate native code to perform I/O requests. Each message sent to an I/O object can be viewed as one atomic step carried out by the I/O object. This can also be thought of as an I/O object processing messages it receives one at a time.

The intuition behind a fair I/O model is that I/O objects cannot be captured by any task. I/O objects always handle requests from tasks in a strict fair way. No task is guaranteed to get two I/O operations processed consecutively and I/O operations of different tasks are arbitrarily interleaved.

Here we use a simple example to demonstrate how the fair I/O model works. `stdin` in the code segment in upper part of Figure 2.9 is the I/O object representing standard input. A task t running the code sends two `=>` messages to `stdin`. If another task t' is also executing the same code; the `stdin` object would receive two `=>` messages from t' as well. As shown in lower part of Figure 2.9, each of the four messages fetches a value from standard input in a separate subtask. So the two reads of t can interleave with those of t' arbitrarily. Neither t nor t' is able to ensure that their two reads from `stdin` are two consecutive inputs.

```
            ...
(L1)        v1 = stdin⇒read();
(L2)        v2 = stdin⇒read();
            ...
```

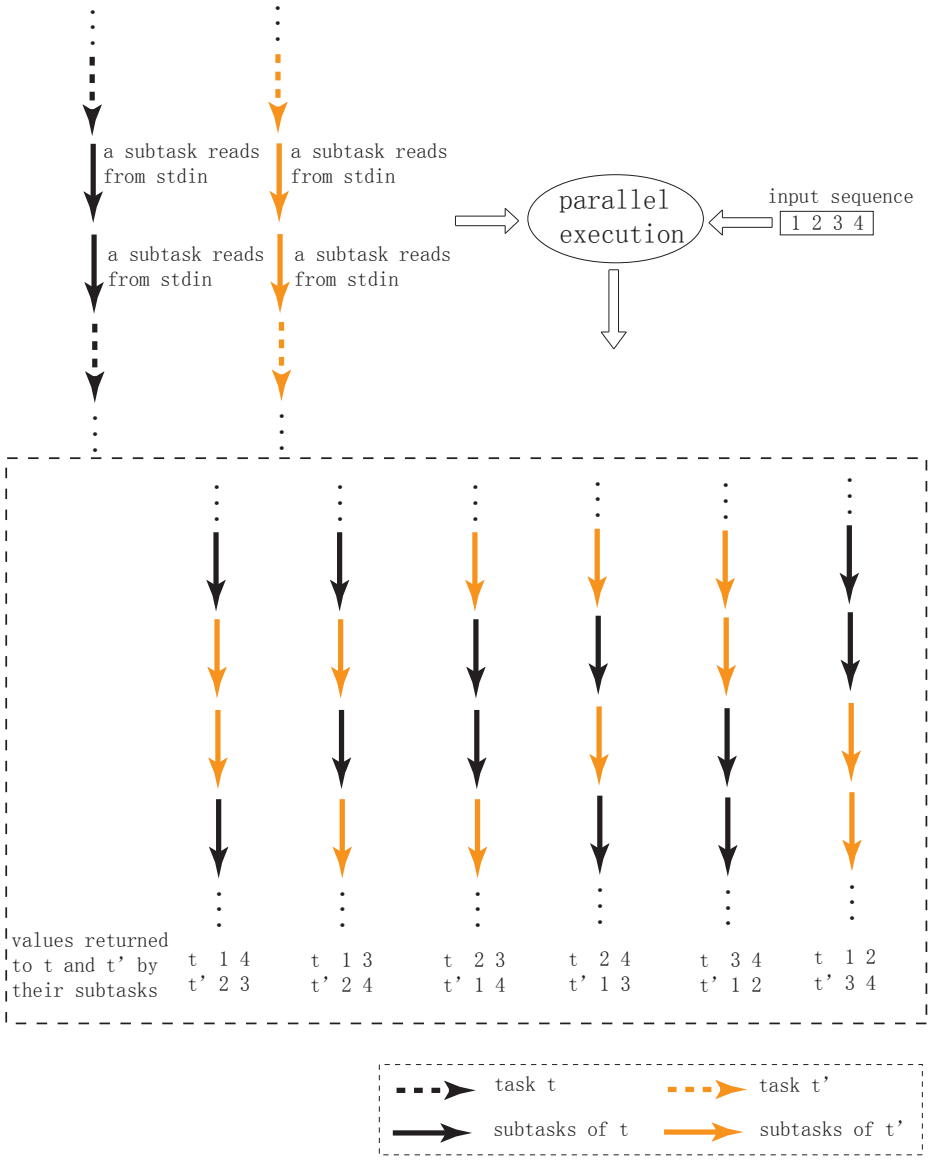

Figure 2.9: A Fair I/O Example

The fair I/O model is the model adopted by all existing languages. For example, the Java `System.out` object is accessible by all Java applications and no thread can lock `System.out` for its exclusive use. If multiple Java threads all output data to `System.out`, these outputs are arbitrarily interleaved, which is exactly the fair I/O model described here. Such an I/O model captures certain common I/O usage patterns. For instance, it is common for a server to log system events into one log file without the concern for interleavings.

Quantized atomicity holds in the fair I/O model: every I/O operation is performed by a new task and each task is a quantum because of the assumption that I/O operations are atomic . However, the fair I/O model is limited because it is impossible for a task to have bigger atomic zones consisting of multiple I/O operations in a single zone. An oversimplified example to demonstrate such a demand would be: an I/O sequence consisting of displaying a question on the screen, getting an input from a user and then replying the user on the screen should naturally be able to be executed without interleavings. In order to achieve atomicity on such an I/O sequence, programmers have to code mutual exclusion manually in the fair I/O model. Again, when we have to rely on programmers programming ability to ensure atomicity, we fall into all the pitfalls we have discussed in Chapter 1.

Until multiple I/O operations can appear in one quantum, the atomic unit in quantized atomicity, it is impossible for a task to access an I/O object in a mutually exclusive way. In the following section, we will discuss how the exact goal is achieved in the reserveable I/O model.

2.3.2 Reserveable I/O

Another way Coqa models I/O is to treat I/O objects more or less the same as common heap objects so they can receive all three types of message send. In such a model, when an I/O object receives a local synchronous (.) message, it is captured by the requesting task exclusively. So a task can request exclusive ownership of an I/O object for the duration of its execution, a stronger atomicity property which allows a task to group multiple I/O operations in some quanta. For instance, the code snippet in Figure 2.10 includes two reads from the `stdin` object via local synchronous messages.

A task t executing the snippet can always get two consecutive inputs from standard input because with the first (.) message, t captures the `stdin` object exclusively until it finishes, which guarantees there is no input between v_1 and v_2 that might have been stolen by other tasks. In other words, the two read I/O operations are in one quantum of t. The lower part of Figure 2.10 shows a concurrent execution of two tasks running the same code. Each of them is able to perform the two reads without interleaving. The reserveable I/O model is very useful in practice. For instance, when a task reads a word, character by character from an I/O object, say the keyboard, it is desirable to read the complete word without missing any characters. In the reserveable I/O model, this can be easily fulfilled by invoking the keyboard I/O object via local synchronous (.) messages. Achieving the same goal in Java on the other hand completely depends on principled programming. Programmers have to make sure by themselves that no tasks other than the task reading from the keyboard are accessing the I/O object. Even so, there is no guarantee that other tasks never intentionally or accidentally steal an input from the keyboard I/O object because I/O objects in Java

```
            ...
(L1)        v1 = stdin.read();
(L2)        v2 = stdin.read();
            ...
```

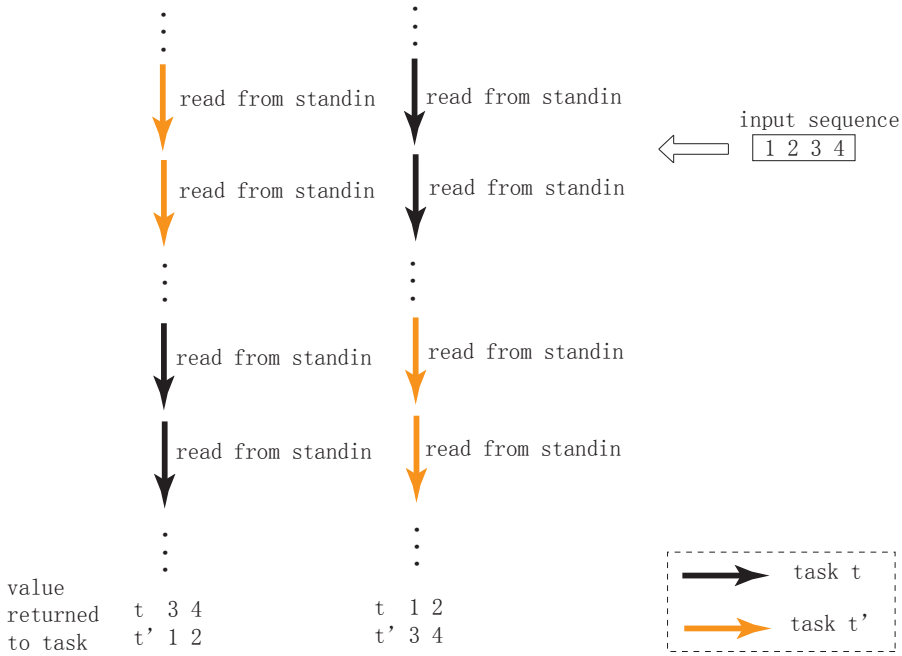

Figure 2.10: A Reserveable I/O Example

are accessible to everyone.

Quantized atomicity holds in the reserveable I/O model because it differs from the orignal Coqa model only on having I/O objects. Because I/O objects can be treated as a special type of common heap objects, introducing them into the Coqa does not break any properties that Coqa has before. The reserveable I/O model has a stronger atomicity property than the fair I/O model in the sense that a task can have atomicity on multiple accesses on an I/O object.

2.3.3 Linkage between I/O Objects

Unlike other heap objects, messages sent to these I/O objects are always globally visible. So I/O objects play a more subtle role in atomicity. Consider the following simple example. Suppose that an application has two tasks t_1 and t_2 accessing two different objects o_1 and o_2, and only these two objects. No matter how steps of t_1 and t_2 interleave, they can be serialized to be either t_1 in its totality happening before t_2 or the other way around. This means that the order of t_1 accessing o_1 and t_2 accessing o_2 does not affect the atomicity of t_1 and t_2.

However, if o_1 and o_2 are two I/O objects representing the two displays which t_1 and t_2 output messages to, and there is an observer sits in front of the monitors all the time, then he can observe the order of the two messages sent to o_1 and o_2 appearing on the two monitors. In this case, serializing steps of t_1 and t_2 has to respect the order of their accesses on o_1 and o_2: if t_1 sends the message to o_1 before t_2 accesses o_2, t_1 has to be serialized so that it happens before t_2, and vice versa. Consider another scenario in which the observer checks the output to monitors only after t_1 and t_2 finish. Then, the order of how t_1 and

t_2 should be serialized becomes arbitrary again. The root of this problem is the linkage between I/O objects. In the second case the ordering of outputs to the two monitors are internal to the application, but in the first case, the observer links o_1 and o_2 and makes previously invisible internal message ordering observable. *Linkage* between two I/O objects can be any kind of connection that associates the two I/O objects such that the ordering of messages sent to the two objects becomes visible to the outside.

In general, I/O objects in the reserveable I/O model are assumed to have no linkage between them. In practice, most I/O objects are indeed independent of each other. For instance, standard input/output can be successfully modeled as I/O objects without any linkages to other I/O objects in most programming language systems.

2.3.4 Bigger Atomic Region: Joinable Quanta

As we have discussed in Section 2.1.4, task creations demarcate quanta. However, in some cases, quanta that are demarcated by task creation can be joined to form a compound quantum which is in fact a bigger atomic region, a union of all constituent quanta.

Consider the example in Figure 2.11. The bootstrapping task t_{main} starts from the special `main` method, and it consists of two quanta, qt_1 from line `L1` to line `L2` and qt_2 of `L4`. The two quanta are delimited by line `L3`, a task creation for spawning task t. Besides the fact that t is a child task of t_{main}, t_{main} and t are independent of each other because they do not have any shared object. It is easy to see that no matter how steps of t_{main} and t interleave, t_{main} is always atomic in its totality. This demonstrates a notion of *joinable quanta*: the two quanta of t_{main}, qt_1 and qt_2, can always join together to form a

larger atomic region if the child task t shares no object with it.

```
        public class Main {
            public static void main (String [] args)  {
(L1)            Foo foo= new foo();
(L2)            int i = 1;
(L3)            foo -> compute(300);
(L4)            i++ ;
          }
        }

        public class Foo {
            public void compute(int v)  {
                 int result = v * v ;
            }
        }
```

Figure 2.11: Joinable Quanta

In Chapter 4 we will present a formal system in which we rigorously prove that certain form of quanta demarcated by task creation can be serialized to be adjacent to each other to form a larger atomic region. Larger atomic regions are better because there would be less interleavings, and hence accomplish a stronger atomicity property.

Chapter 3

KernelCoqa: the Coqa 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

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

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

Figure 3.1: Language Abstract Syntax

$H\{x \mapsto y\}$ 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.

The abstract syntax of our system is shown in Figure 3.1. 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 `=>` and `->`, respectively. A class declared `exclusive` will have its objects write captured upon any access. This label is 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.

3.2 Operational Semantics

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	::=	$o \mid \textbf{null}$	*current executing object*
v	::=	$cst \mid o \mid \textbf{null}$	*values*
e	::=	$v \mid \textbf{wait}\ t$	*extended expression*
		$\mid e \uparrow e \mid \ldots$	
$\mathbf{E}$	::=	$\bullet \mid fn = \mathbf{E}$	*object evaluation context*
		$\mid \mathbf{E}.m(e) \mid v.m(\mathbf{E})$	
		$\mid \mathbf{E} \rightarrow m(e) \mid v \rightarrow m(\mathbf{E})$	
		$\mid \mathbf{E} \Rightarrow m(e) \mid v \Rightarrow m(\mathbf{E})$	
		$\mid \textbf{let}\ x = \mathbf{E}\ \textbf{in}\ e$	
o			*object ID*
t			*task ID*

Figure 3.2: Dynamic Data Structures

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 the set of parallel tasks. Every task in turn has its local evaluation context $\mathbf{E}$. The relevant definitions are given in Figure 3.2. 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

$$\textsc{TContext1}\quad \frac{H, N, T_1 \Rightarrow H', N', T_1'}{H, N, T_1 \parallel T_2 \Rightarrow H', N', T_1' \parallel T_2} \qquad \textsc{TContext2}\quad \frac{H, N, T_2 \Rightarrow H', N', T_2'}{H, N, T_1 \parallel T_2 \Rightarrow H', N', T_1 \parallel T_2'}$$

Figure 3.3: KernelCoqa Operational Semantics Rules (1)

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 run time expressions **wait** t and $e \uparrow e$.

Figure 3.3 shows the concurrent context for parallel tasks. The complete single-step evaluation rules for each task are presented in Figure 3.4. In this presentation, we use $e; e'$ as shorthand for **let** $x = e$ **in** e' where x is a fresh variable. These rules are implicitly defined over some fixed program P. Every rule in Figure 3.4 has a label after the rule name. These labels are used in subsequent definitions and lemmas for proving quantized atomicity in quantized atomicity.

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 creating task. It may look like that the message mn is just re-sent 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'. For simplicity, asynchronous messages sent to an object are handled in a non-deterministic order. However, it is not technically difficult to enforce the processing order of asynchronous messages sent from a

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

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

$$\textsc{This}(t,\gamma)$$
$$H,N,\langle t;\gamma;\mathbf{E}[\,\mathbf{this}\,]\rangle\Rightarrow H,N,\langle t;\gamma;\mathbf{E}[\,\gamma\,]\rangle$$

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

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

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

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

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

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

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

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

Figure 3.4: KernelCoqa Operational Semantics Rules (2)

sender object to a receiver object to be the sending order of these messages.

The SubTask rule creates a subtask of the current task via synchronous messaging. The creating task (the parent) goes in **wait** 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:

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

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.

The Set rule specifies that a task t can write capture object γ, the current object t is operating on, if all write capturers and 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 Section 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. In reality, fairness comes at a significant performance cost[26]. The bookkeeping and synchronization required to ensure fairness means that contended fair locks will have much lower throughput than unfair locks. In most cases, a statistical fairness in which a blocked thread will eventually proceed is often good enough, and the performance benefits of nonfair locks outweigh the benefits of fair ones.

Other rules in Figure 3.4 are standard. For instance, for simplicity reason, object constructors in KernelCoqa are no much different than common methods except having the class names and they take no arguments. Accordingly object instantiation rule INST creates a new object, initializes all of its 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. The key Lemma is the Bubble-Down Lemma, Lemma 2, which shows that consecutive steps of a certain form in a computation

path can be swapped to give an "equivalent" computation path, a path with the same computational behavior. Then, by a series of bubblings, steps of each quantum can be bubbled to all be consecutive 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 every *pmsp* in the path is serializable and thus the whole path is serializable in unit of *pmsp*'s.

Definition 1 (Object State)**.** *Recall the global state is a triple* $S = (H, N, T)$*. H is the object heap, N a task ancestry mapping, and T the set of parallel tasks. The* object state *for o, written s_o, is defined as $H(o) = \langle cn; R; W; F\{fn \mapsto v\}\rangle$, the value of the object o in the current heap H, or* **null** *if $o \notin \mathsf{dom}(H)$.*

We write step $st_r = (S, r, S')$ to denote a transition $S \Rightarrow S'$ by rule r of Figure 3.4. We let $change(st_r) = (s_o, r, s'_o)$ denote the fact that the begin and end heaps of step st_r differ at most on their state of o, taking it from s_o to s'_o. Similarly, the change in two consecutive steps which changes at most two objects o_1 and o_2, $o_1 \neq o_2$, is represented as $change(st_{r_1} st_{r_2}) = ((s_{o_1}, s_{o_2}), r_1 r_2, (s'_{o_1}, s'_{o_2}))$. If $o_1 = o2$, then $change(st_{r_1} st_{r_2}) = (s_{o_1}, r_1 r_2, s'_{o_1})$.

Definition 2 (Local and Nonlocal Step)**.** *A step $st_r = (S, r, S')$ is a* local step *if r is one of the local rules: either* Get, Set, This, Let, Return, Inst *or* Invoke*. st_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 3.4. For example, the Task rule has

label $\textsc{Task}(t, \gamma, mn, v, o, t')$ 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 also carry labels but since they are local steps internal to a t so that they are *not* observable steps. However, when I/O is introduced to KernelCoqa in Chapter 4, some local steps that perform I/O operations would become external visible steps, until then, we do not care about labels attached to these local rules.

Intuitively, observable steps are places where tasks interact by making results of local steps possibly visible to other tasks. For instance, in a Task step, a task t creates a new task t' by sending an asynchronous message mn. Along with the message mn, t sends a parameter v to t'. If v is a value calculated by t's local steps, then upon the execution of the Task step, t makes its internal computation visible to the newly created t'.

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

Proof. A local step is step of applying one of the local rules defined in Definition 2. Among all these rules, only Inst, Get and Set can possibly change object state. Inst creates a single object. The Get and Set rules each operates on exactly one object and may change is its state. No other rules change object state. □

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

$st_{r_1} st_{r_2} \ldots st_{r_{i-1}} st_{r_i} = (S_0, r_1, S_1)(S_1, r_2, S_2) \ldots (S_{i-2}, r_{i-1}, S_{i-1})(S_{i-1}, r_i, S_i)$.

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.

For brevity purpose, we use computation path and path in an interchangeable way if no confusion would arise.

The initial run time configuration of a computation path, S_0, always has $H = \emptyset$, $N = \emptyset$ and $T = \{t_{main}\}$. t_{main} is the default bootstrapping task that starts the execution of a program from the entry `main` method of a class in the program.

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

Note that this definition can encompass 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. We call such an I/O model the global I/O model and details about this model is covered in Section 4.1.

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

Object-blocked state is a state a task cannot make process because it has to wait for an object becoming available to it.

Definition 7 (Sub-path and Maximal Sub-path)**.** *Given a fixed p, for some task 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 consecutive 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 a task 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 st_r of p occurs in precisely one pmsp, written as $st_r \in pmsp$.*

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

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

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

Definition 9 defines a global ordering on all *pmsp*'s in a path p without concerning which task a *pmsp* belongs to. The following Definition 11 defines a task scope index of a *pmsp* which is used to indicate the local ordering of *pmsp*'s of a task within the scope of the task.

Definition 11 (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* $pmsp_{t,i-1}$*.*

For a $pmsp_{t,i}$, if we do not care its task scope ordering, we omit the index i and simply use $pmsp_t$.

Definition 12 (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 13 (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 the computation. Use of such a Lemma is the standard technique to show atomicity properties. Lipton [44] first described such a theory, called *reduction*; his theory was later refined by [41]. In this approach, all state transition steps of a potentially interleaving execution are categorized based on their commutativity with consecutive 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 [23, 25] 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.

Definition 14 (Step Swap)**.** *For any two consecutive steps $st_{r_1} st_{r_2}$ in a computation path p, a* step swap *of $st_{r_1} st_{r_2}$ is defined as swapping the order of application of rules in the two steps, i.e., apply r_2 first then r_1. We let $st'_{r_2} st'_{r_1}$ denote a step swap of $st_{r_1} st_{r_2}$.*

Definition 15 (Equivalent Step Swap)**.** *For two consecutive steps $st_{r_1} st_{r_2}$ in a computation 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 2 (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 st_{r_1} is a local step, then a step swap of $st_{r_1} st_{r_2}$ is an equivalent step swap.*

Proof. First, observe that if t_2 is a subtask of t_1, then it is impossible for st_{r_1} to be a local step while st_{r_2} is a step of t_2. Because according to semantics defined in Figure 3.4 a task

and its subtask never have their local steps consecutive to each other. Figure. 3.5 illustrates all possible cases of how steps of a task and steps of its subtask may layout in a path. It clearly shows that local steps of the task t_1 and its subtask t_2 always have their local steps demarcated by some nonlocal steps.

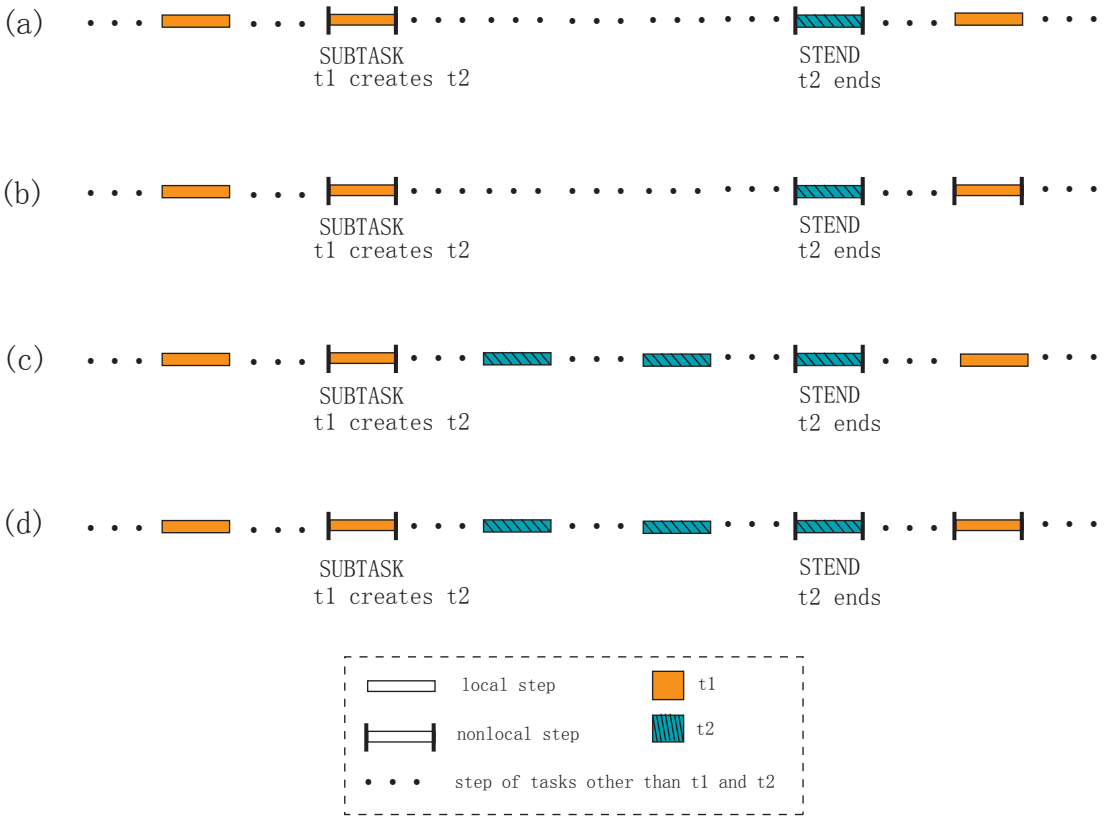

Figure 3.5: Task and Subtask

According to Definition 4 and 5, $\equiv$ is defined by the sequence of labels of nonlocal steps occurring in path p, so a step swap of $st_{r_1} st_{r_2}$ always gives a new path p', $p' \equiv p$, since st_{r_1} is a local step by the lemma so that swap st_{r_1} with any st_{r_2} never changes the ordering

of labels. Therefore, to show that the step swap of $st_{r_1} st_{r_2}$ is an equivalent step swap we only need to prove that if $st_{r_1} st_{r_2} = (S, r_1, S')(S', r_2, S'')$, then the step swap of $st_{r_1} st_{r_2}$ is $st'_{r_2} st'_{r_1} = (S, r_2, S^*)(S^*, r_1, S'')$.

Because st_{r_1} is a local step, it can at most change one object's state on the heap H and a local step does not change N according to the operational semantics. So we can represent st_{r_1} as $st_{r_1} = ((H_1, N_1, T_1), r_1, (H_2, N_1, T_2))$ where H_1 and H_2 differ at most on one object o.

(Case I) st_{r_2} is also a local step.

So st_{r_2} does not change N either, so $st_{r_2} = ((H_2, N_1, T_2), r_2, (H_3, N_1, T_3))$. Because st_{r_1} and st_{r_2} are steps of different tasks t_1 and t_2, st_{r_1} and st_{r_2} must change different elements of T (t_1 and t_2 respectively) by inspection of the rules. This means any change made by st_{r_1} and st_{r_2} to T always commute. So in the section of **Case I**, we focus only on changes st_{r_1} and st_{r_2} make on H, and omit N and T for concision.

Suppose $change(st_{r_1}) = (s_{o_1}, r_1, s'_{o_1})$ and $change(st_{r_2}) = (s_{o_2}, r_2, s'_{o_2})$.

If $o_1 \neq o_2$, then $change(st_{r_1} st_{r_2}) = ((s_{o_1}, s_{o_2}), r_1 r_2, (s'_{o_1}, s'_{o_2}))$. Swapping the order of st_{r_1}, st_{r_2} by applying r_2 first then r_1 results in the change $((s_{o_1}, s_{o_2}), r_2 r_1, (s'_{o_1}, s'_{o_2}))$, which has the same start and end states as $change(st_{r_1} st_{r_2})$, regardless of what r_1 and r_2 are.

If $o_1 = o_2 = o$, then $change(st_{r_1}) = (s_o, r_1, s'_o)$, $change(st_{r_2}) = (s'_o, r_2, s''_o)$ and $change(st_{r_1} st_{r_2}) = (s_o, r_1 r_2, s''_o)$. Let $s_o = \langle cn; R; W; F \rangle$.

By inspection of the rules this case only arises if both of the rules are amongst GET, SET and INST.

Subcase a: $r_1 =$ INST.

$change(st_{r_1}) = (\textbf{null}, r_1, s_o)$ so r_2 cannot be INST since o cannot be created twice. And, r_2 also cannot be any other local rule: o is just created in st_{r_1} by t_1. For t_2 to be able to access o, it must obtain a reference to o first. Only t_1 can pass a reference of o to t_2 directly or indirectly. As a result, if st_{r_2} is a local step of t_2 that operates on o, it cannot be consecutive to st_{r_1}, and vice visa. Therefore, r_1 INST is not possible.

Subcase b: $r_1 =$ GET.

Trivially, r_2 cannot be INST that creates o because no steps can operate on an object before it is created.

If the class of o is declared as `exclusive`, then $change(st_{r_1}) = (\langle cn; R; W; F\rangle, r_1, \langle cn; R; W'; F\rangle)$, $W' = W \cup \{t\}$ if t_1 captures o in st_{r_1} or $W' = W$ if W is a subset of the ancestors of t_1. Either way, there does not exist an consecutive st_{r_2} of t_2 where r_2 is either GET or SET because if so, firing up st_{r_2} would violate the preconditions of the GET or the SET rule.

If the class of o is not declared `exclusive`, then $change(st_{r_1}) = (\langle cn; R; W; F\rangle, r_1, \langle cn; R \cup \{t_1\}; W; F\rangle)$. If $W \neq \emptyset$, then there does not exist a st_{r_2} of t_2 where r_2 is either GET or SET because if so st_{r_2} would violate the precondition of the GET or the SET rule. In the case of $W = \emptyset$, r_2 can only be GET on o because if it is SET, firing up st_{r_2} would violate the precondition of SET. In this case, $change(st_{r_1} st_{r_2}) = (\langle cn; R; W; F\rangle, r_1 r_2, \langle cn; R \cup \{t_1\} \cup \{t_2\}; W; F\rangle)$. Swapping the application order of r_1 and r_2 we get $change(st'_{r_2} st'_{r_1}) = (\langle cn; R; W; F\rangle, r_2 r_1, \langle cn; R' \cup \{t_2\} \cup \{t_1\}; W; F\rangle)$, which is the same as $change(st_{r_1} st_{r_2})$ because set union commutes.

Subcase c: $r_1 =$ SET.

Trivially, r_2 cannot be INST that creates o because no steps can operate on an object before it is created.

Let $st_{r_1} = (\langle cn; R; W; F\rangle, r_1, \langle cn; R; W'; F'\rangle)$ and $t_1 \in W'$. Then r_2 cannot be GET or SET because if so st_{r_2} would violate the precondition of the GET or the SET rule. So r_1 cannot be SET

(Case II) st_{r_2} is a nonlocal step.

Let $st_{r_1} = ((H_1, N_1, T_1), r_1, (H_2, N_1, T_2))$ and H_2 differs from H_1 at most on object o since st_{r_1} is a local step. Because st_{r_1} and st_{r_2} are steps of t_1 and t_2 respectively, they must change different elements of T, i.e., t_1 and t_2. st_{r_2} as a nonlocal step may add a new fresh element t to T. But st_{r_1} and st_{r_2} still obviously commute in terms of changes to T. So in the following proof, we do not need to concern about T commutativity.

Subcase a: $r_2 =$ TASK.

Let t be the new task created in st_{r_2}, then $st_{r_2} = ((H_2, N_1, T_2), r_2, (H_2, N_1, T_2 \cup t))$. The final state change of taking st_{r_1} and st_{r_2} in this order is $((H_1, N_1, T_1), r_1 r_2, (H_2, N_1, T_2 \cup t))$. Swapping st_{r_1} and st_{r_2} results consecutive steps $st'_1 = ((H_1, N_1, T_1), r_2, (H_1, N_1, T_1 \cup t))$ and $st'_2 = ((H_1, N_1, T_1 \cup t), r_1, (H_2, N_1, T_2 \cup t))$, which makes the combined state change of $st'_1 st'_2$ to be $((H_1, N_1, T_1), r_2 r_1, (H_2, N_1, T_2 \cup t))$, the same state change as that of $st_{r_1} st_{r_2}$.

Subcase b: $r_2 =$ SUBTASK.

Let t be the new task created in st_{r_2}, then $st_{r_2} = ((H_2, N_1, T_2), r_2, (H_2, N_1 \cup \{t_2 \mapsto t\}, T_3))$. $st_{r_1} st_{r_2} = ((H_1, N_1, T_1), r_1 r_2, (H_2, N_1 \cup \{t_2 \mapsto t\}, T_3))$. If we apply r_2 first, we get $st'_1 = ((H_1, N_1, T_1), r_2, (H_1, N_1 \cup \{t_2 \mapsto t\}, T_2))$. Then r_1 is applied to get $st'_2 = ((H_1, N_1 \cup \{t_2 \mapsto t\}, T_2), r_1, (H_2, N_1 \cup \{t_2 \mapsto t\}, T_3))$. Therefore, $st'_1 st'_2$ results in a transition

$((H_1, N_1, T_1), r_2 r_1, (H_2, N_1 \cup \{t_2 \mapsto t\}, T_3)$ which has the same start and final states as $st_{r_1} st_{r_2}$.

Subcase c: $r_2 = \text{TEND}$.

Let $st_{r_2} = ((H_2, N_1, T_2), r_2, (H_3, N_1, T_3))$, where

$H_3 = \biguplus_{H_2(o)=\langle cn;R;W;F\rangle} (o \mapsto \langle cn; R\backslash t_2; W\backslash t_2; F\rangle))$. Namely, st_{r_2} removes t_2 from R and W sets of all objects in H_2. Consider the case when st_{r_1} changes an object o's F but not its R or W: taking st_{r_1} then st_{r_2} has the same state change as taking the two steps in reversed order because the two steps work on different regions of the heap. If st_{r_1} also changes R or W of o, then it can only add t_1 to R or W of o, while st_{r_2} may only remove t_2 from R or W of o. Consequently, swapping the two steps we still get the same result because the set operations commute since $t_1 \neq t_2$.

Subcase d: $r_2 = \text{STEND}$.

Let $N(t) \mapsto t_2$, then $st_{r_2} = ((H_2, N_1, T_2), r_2, (H_3, N_1\backslash t, T_3))$, where

$H_3 = \biguplus_{H_2(o)=\langle cn;R;W;F\rangle} (o \mapsto \langle cn; R\backslash t_2; W\backslash t_2; F\rangle)$. Namely, st_{r_2} removes t_2 from R and W sets of all objects in H_2. Consider the case when st_{r_1} changes an object o's F but not its R or W: taking st_{r_1} then st_{r_2} has the same state change as taking the two steps in reversed order because the two steps work on different regions of the heap. If st_{r_1} adds t_1 to R or W of o, swapping the two steps we still get the same result because the set operations commute since $t_1 \neq t_2$. □

Given this Lemma we can now directly prove the Quantized Atomicity Theorem.

Theorem 1 (Quantized Atomicity)**.** *For all paths p there exists a quantized path p' such that $p' \equiv p$.*

Proof. Proceed by first sorting all *pmsp*'s of p into a well ordering induced by the ordering of their points in p. Write $pmsp(i)$ for the i-th indexed *pmsp* in this ordering. Suppose that there are n *pmsp*'s in total in p for some n. We proceed by induction on n to show for all $i \leq n$, p is equivalent to a path p_i where the $1st$ to ith indexed *pmsp*'s in this ordering have been bubbled to be quantized subpaths in a prefix of p_i: $p \equiv p_i = pmsp(1) \dots pmsp(i) \dots$ where $pmsp(k)$ is quantized with $k = 1 \dots i$. With this fact, for $i = n$ we have $p \equiv p_n = pmsp(1) \dots pmsp(n)$ where $pmsp(k)$ is quantized with $k = 1 \dots n$, proving the result.

The base case $n = 0$ is trivial since the path is empty. Assume by induction that all $pmsp(i)$ for $i < n$ have been bubbled to be quantized subpaths and the bubbled path $p_i = pmsp(1) \dots pmsp(i) \dots$ where $pmsp(k)$ is quantized with $k = 1 \dots i$, has the property $p_i \equiv p$. Then, the path around $pmsp(i+1)$ includes steps of $pmsp(i+1)$ or *pmsp*'s with bigger indices. By repeated applications of the Bubble-Down Lemma, all these local steps that do not belong to $pmsp(i+1)$ can be pushed down past its point, defining a new path p_{i+1}. In this path $pmsp(i+1)$ is also now a quantized subpath, and $p_{i+1} \equiv p$ because $p_i \equiv p$ and the Bubble-Down lemma which turns p_i to p_{i+1} does not shuffle any nonlocal steps so $p_i \equiv p_{i+1}$. □

Figure 3.6 demonstrates the proof of Theorem 1 *Quantized Atomicity*. In this illustration, t_1 and t_2 are two tasks and t_1 creates t_2. Once t_2 is started, it executes with t_1 in an interleaved way. Figure 3.6(a) shows one instance of execution paths which have interleaved steps of t_1 and t_2. `TASK` is the step t_1 sends an asynchronous `->` message that creates t_2 and the two `TEND` steps are the last steps of t_1 and t_2 respectively. To proceed with the proof of the *Quantized Atomicity*, we first need to identify and index all *pmsp*'s in

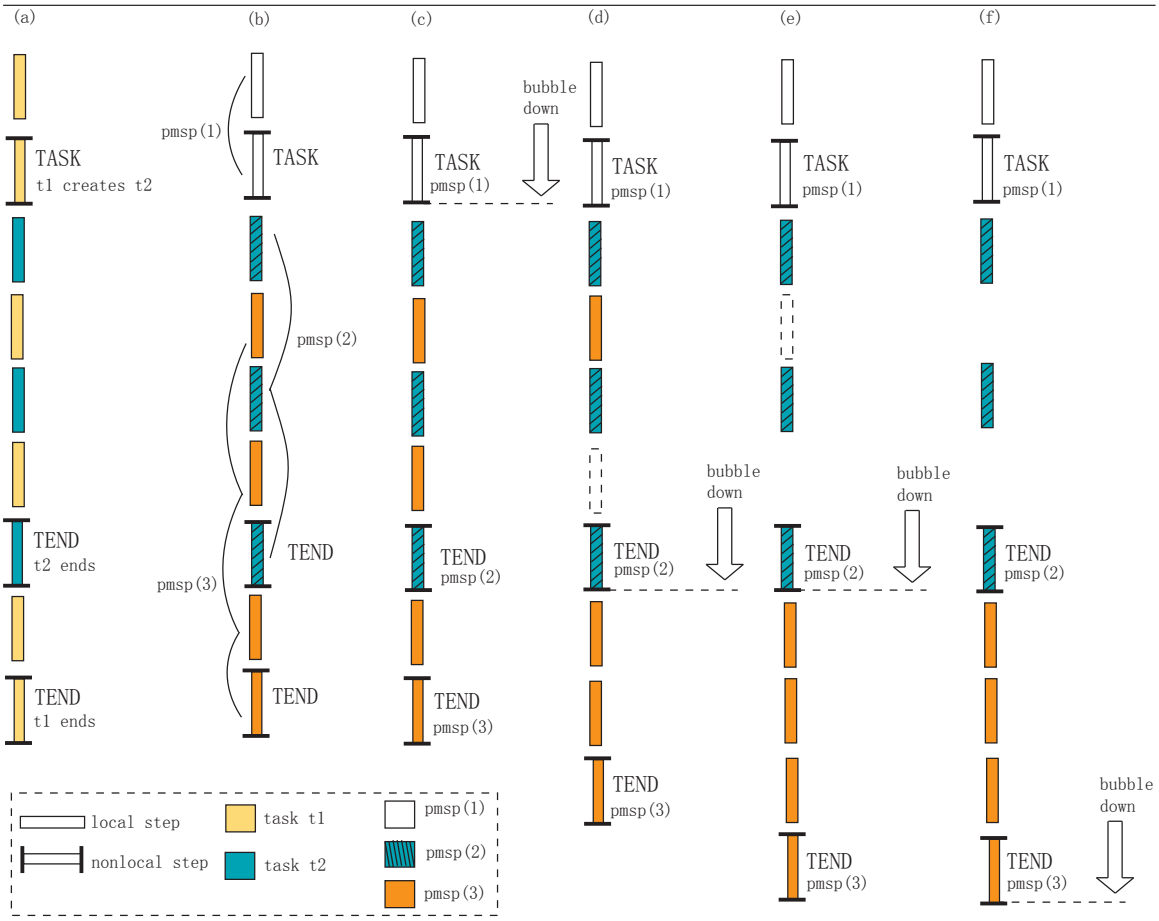

Figure 3.6: Quantized Atomicity in KernelCoqa

the execution path, as shown in Figure 3.6(b). Each *pmsp* has a nonlocal step at the end. These *pmsp*'s are indexed by their ordering of their nonlocal steps in the execution path. In this example, t_1 has two *pmsp*'s, *pmsp*(1) and *pmsp*(3), while t_2 has only one *pmsp*(2). Figure 3.6(c)-(f) show how the three *pmsp*'s are serialized by using the Bubble-down Lemma repeatedly in the order of the indexed *pmsp*'s. First, *pmsp*(1) is already serialzied in the sense that there is no step that does not belong to this *pmsp* interleaving with the steps of *pmsp*(1). Then, we can proceed to the second *pmsp*, *pmsp*(2). As shown in Figure 3.6(d) and (e), applying the Bubble-down lemma to *pmsp*(2) requires two step swaps, which results the *pmsp*(2) to be sequentialized in (e). At last, the Bubble-down lemma is applied to *pmsp*(3), which is a no-op since *pmsp*(3) becomes sequentialized after the step swaps for *pmsp*(2). Finally, we have a quantized path in (f) that is observably equivalent to the original path in (a). According to the proof, we can conclude that $t1$ has a quantized atomicity consisting of two atomic quanta. $t2$ has one atomic quantum.

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 consecutive local steps, where at least one of the two accesses changes the value of the field.*

Proof. First, the two tasks cannot be related by ancestry because subtasks never run in parallel with tasks. To update an object field, a task needs to obtain a write lock which prevents other tasks from reading or writing the same object. When a field is read by a task, the task needs to be granted a read lock first which prevents other tasks from writing

the same object in the next step. Thus if one of the steps is a write of one task, the other step cannot be a read or write of an unrelated task. □

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.*

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.

Chapter 4

Building I/O in KernelCoqa

Section 2.3 gives an informal description of how Coqa models I/O in two different ways: the fair I/O model and the reserveable I/O model. Both models use special I/O objects as the abstraction of I/O channels. I/O objects in the two models are different in types of message they can receive. In the fair model, only nonlocal messages (-> and => messages) can be sent to an I/O object, while in the reserveable model, an I/O object can handle all types of messages including local (.) messages.

In this chapter, we formally establish the I/O two models as extensions to KernelCoqa and we call them KernelCoqa$_{fio}$ and KernelCoqa$_{rio}$ respectively. The focus of this chapter is to study I/O atomicity and implications an I/O module brings to the whole language system, which have not yet been thoroughly explored in other systems.

KernelCoqa$_{fio}$ and KernelCoqa$_{rio}$ redefine observable behaviors in terms of I/O objects because with introducing of I/O, systems in interaction with KernelCoqa are independent to each other with respect to I/O in the sense that each system can only observe

the peer system's I/O actions but not its internal computation. So from other systems' point of view, all it matters in terms of KernelCoqa's behavior is its I/O behavior. New results then are proved based on the new definitions of I/O behaviors in KernelCoqa$_{fio}$ and KernelCoqa$_{rio}$ respectively.

Similar to the proof techniques used in Section 3.3, the key in proving theoretical results in KernelCoqa$_{fio}$ and KernelCoqa$_{rio}$ is to show 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. The *pmsp*, the technical notion of a quantum, is still the unit of path serialization. Like KernelCoqa, KernelCoqa$_{fio}$ systematically serializes each *pmsp* in a computation path. Once every *pmsp* of the given computation path is serialize, the whole path is then quantized. KernelCoqa$_{rio}$ first shows that every computation path has an equivalent quantized path that satisfies the I/O equivalence relationship defined in KernelCoqa$_{rio}$, then we prove lemmas which allow certain *pmsp*'s in an already quantized path to be shuffled within the path without affecting the path's I/O observable behavior. Consequently, certain *pmsp*'s of a task can be moved to be adjacent to each other, forming a bigger atomic region.

4.1 Fair I/O Model

In the fair I/O model, I/O is manifested on some fixed objects. Specifically, KernelCoqa$_{fio}$ has a built-in set, denoted by $SysIO$, of pre-defined I/O objects. Each object in $SysIO$ represents an I/O channel between the KernelCoqa$_{fio}$ system and the world outside. For instance, there is one object associated with standard input and one

$$\frac{\textsc{Task-IO}(t, \gamma, mn, v, o) \qquad o \in SysIO}{H, N, \langle t; \gamma; \mathbf{E}[\, o \twoheadrightarrow mn(v)\,]\rangle \Rightarrow H, N, \langle t; \gamma; \mathbf{E}[\,\mathbf{null}\,]\rangle}$$

$$\frac{\textsc{SubTask-IO}(t, \gamma, mn, v, v', \gamma) \qquad o \in SysIO}{H, N, \langle t; \gamma; \mathbf{E}[\, o \Rightarrow mn(v)\,]\rangle \Rightarrow H, N, \langle t; \gamma; \mathbf{E}[\, v'\,]\rangle}$$

Figure 4.1: Operational Semantics Rules for I/O Objects in KernelCoqa$_{fio}$

with standard output and each file/device is represented by one I/O object. I/O objects encapsulate native code for channeling data in/out the KernelCoqa$_{fio}$ system, so messages sent to/by them are globally visible system input/output.

KernelCoqa$_{fio}$ has the same operational semantics as KernelCoqa except the two extra rules for handling I/O messages as shown in Figure 4.1. In the Task-IO rule, object γ in task t sends an asynchronous message to an I/O object o. Notice that there is no new task created as with the Task rule in Figure 3.4. Similarly, no new subtask is created in the SubTask-IO rule. The Task-IO rule specifies that no return value is expected from the I/O object o. The SubTask-IO rule states an arbitrary value v' is fed back from o because an I/O input is arbitrary to the receiving system. There are no task end rules corresponding to the Task-IO and the SubTask-IO rules because no new task is created. To understand this semantics, we can think of the native operations performed by I/O objects are beyond the scope of KernelCoqa$_{fio}$ operational semantics so they are treated as one message send step from KernelCoqa$_{fio}$'s point of view.

We here rigorously prove quantized atomicity in KernelCoqa$_{fio}$ with respect to I/O behavior. First, we introduce a new definition of I/O observable equivalence $\equiv_{fio}$. Then, we show observable equivalence $\equiv$ defined in Definition 5 is a sufficient condition of I/O

observable equivalence $\equiv_{fio}$: $\equiv$ is a subset of $\equiv_{fio}$. We prove this theorem by demonstrating that the I/O behavior is a sub-sequence overlapping with the observable behavior defined in Definition 4.

Definition 16 (Nonlocal Step and I/O-nonlocal Step)**.** *A step st_r is a* nonlocal step *if r is one of the nonlocal rules: either* TASK*,* SUBTASK*,* TEND*,* STEND*,* TASK-IO *or* SUBTASK-IO*. st_r is an* I/O-nonlocal *step if r is either* TASK-IO *or* SUBTASK-IO*.*

The nonlocal step defined above is no difference from the one previously defined by Definition 2 of Section 3.3 except that it distinguishes nonlocal messages sent to I/O objects from those to common objects. Specifically, the Task-IO rule is a sub-case of the TASK rule, a case when an asynchronous message is sent to an I/O object. Similarly, SUBTASK-IO is a special case of SUBTASK when a subtasking message is dispatched to an I/O object.

Definition 17 (I/O Observable Behavior ob_{fio})**.** *The I/O observable behavior of a computation path p, $ob_{fio}(p)$, is the label sequence of all I/O-nonlocal steps occurring in p.*

Definition 18 (I/O Observable Equivalence $\equiv_{fio}$)**.** *Two paths p_1 and p_2 are observably equivalent, written as $p_1 \equiv_{fio} p_2$, iff $ob_{fio}(p_1) = ob_{fio}(p_2)$.*

Lemma 3. *Lemma 2 (Bubble-down Lemma) holds in KernelCoqa$_{fio}$*

Proof. KernelCoqa$_{fio}$ extends the operational semantics of KernelCoqa by two nonlocal rules given in Figure 4.1. To proof that the Bubble-down Lemma still holds in the extended operational semantics, we only need to extend the proof of Lemma 2 by adding the following subcase to **(Case II) st_{r_2} is a nonlocal step**.

Subcase e:

r_2 = Task-IO or SubTask-IO. By inspection of Task-IO and SubTask-IO, st_{r_2} does not change $S = H, N, T$. Obviously, a step swap of $st_{r_1} st_{r_2}$ is an equivalent step swap. □

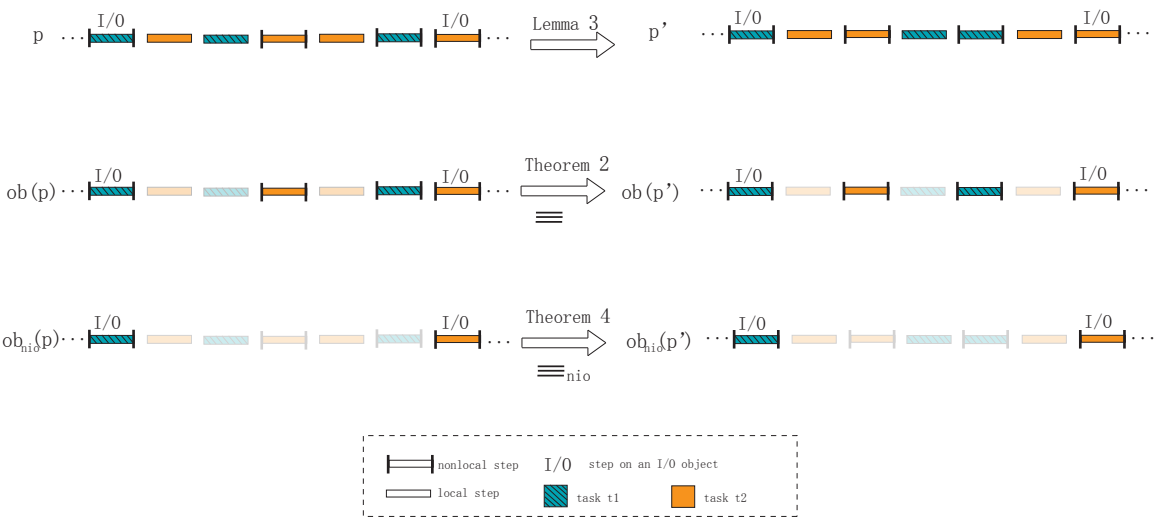

Figure 4.2: Quantized Atomicity in KernelCoqa$_{fio}$

Theorem 4 (Quantized Atomicity with $\equiv_{fio}$)**.** *For all paths p there exists a quantized path p' such that $p' \equiv_{fio} p$.*

Proof. By Lemma 3, Quantized Atomicity Theorem (Theorem 1) holds in KernelCoqa$_{fio}$, so for any path p, there exists a quantized path p' such that $p' \equiv p$. Moreover, all I/O steps are nonlocal steps, so $p' \equiv_{fio} p$ as well. Theorem 4 directly follows Theorem 1. □

Figure 4.2 illustrates how the I/O observable behavior in KernelCoqa$_{fio}$ is subsumed by the observable behavior defined in KernelCoqa so that $p \equiv p'$ implies that

$p \equiv_{fio} p'$.

```
            ...
(L1)        v1 = stdin ⇒ read();
(L2)        v2 = stdin ⇒ read();
            ...
```

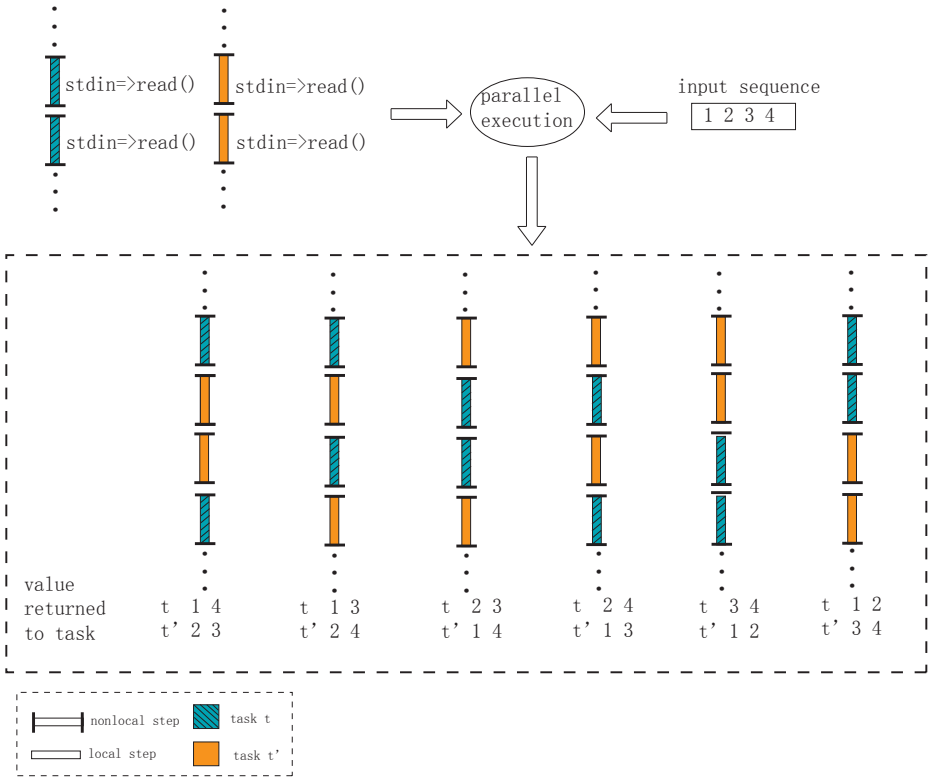

Figure 4.3: An Example in KernelCoqa$_{fio}$

Fact 2. *I/O operations are atomic.*

By inspection of operational semantics rules of KernelCoqa$_{fio}$, every message to an I/O object is performed by exactly one nonlocal step, a nonlocal TASK-IO/SubTask-IO step. So every I/O operation is one atomic operation in KernelCoqa$_{fio}$. Data race freedom

on I/O objects and mutual exclusion over tasks accessing I/O objects directly follow Fact 2.

In KernelCoqa$_{fio}$, it is impossible for a task to have multiple I/O operations to be processed as one atomic operation because the operational semantics of KernelCoqa$_{fio}$ determines that every I/O operation is performed by exactly one nonlocal step so one *pmsp* may include at most one I/O operation.

Figure 4.3 depicts the same example in Figure 2.9: two concurrent tasks t and t' each send two I/O messages to `stdin`, the I/O object representing standard input. But Figure 4.3 more accurately illustrates operational steps according to KernelCoqa$_{fio}$'s semantics. Notice that Figure 4.3 and Figure 2.9 are essentially the same except that steps of spawning a subtask for an I/O message in Figure 2.9 collapse into a single nonlocal step in Figure 4.3.

4.2 Reserveable I/O Model

In this section, we describe KernelCoqa$_{rio}$, the KernelCoqa with a *reserveable* I/O model extension. KernelCoqa$_{rio}$ gives mutual exclusion on I/O channels which allows multiple I/O operations to be grouped into one quantum. This is achieved by allowing I/O objects to receive local synchronous (.) message in addition to nonlocal -> and => messages.

Figure 4.4 shows the special pre-defined I/O class in KernelCoqa$_{rio}$. Every instance of I/O resources corresponds to exactly one I/O object. Because I/O objects can receive local synchronous (.) messages, they can be captured by a task as normal heap objects. Notice that the I/O class is declared with an `exclusive` label, which means an I/O object

```
template exclusive IOClass {
   IOClass() {  }
   byte read() {
      in = rand();
      return in;
   }
   void write(byte v) {
      out = v;
   }
   int random() {
     // ... generate and return a random integer
   }

   byte in;     //input value
   byte out;    //output value
}
```

Figure 4.4: I/O Class in KernelCoqa$_{rio}$

is always write captured upon any access. The `IOClass` in Figure 4.4 is just an abstraction of actual I/O and we write it in a Java-alike syntax for easy reading. The `IOClass` emulates I/O by sinking data sent to it or returning arbitrary random values computed by the `random` method. The two fields `in` and `out` are declared for the purpose of causing an I/O object to be captured by a task. In a real implementation, the `IOClass` would include native code that writes data in and out KernelCoqa$_{rio}$ system which is beyond the scope of the formalization.

Definition 19 (I/O Class and I/O Objects)**.** I/O class *in KernelCoqa$_{rio}$ is a system defined class as shown in Figure 4.4.* I/O objects *are objects on the heap H such that $H(o) = \langle cn; W; R; F\rangle$ where $cn =$* IOClass.

It is easy to see that I/O objects are treated the same as common heap objects

in terms of messaging schemes. So KernelCoqa$_{rio}$ has the same operational semantics as KernelCoqa as defined in Figure 3.4. However, KernelCoqa$_{rio}$ has to take labels of the GET and SET rules into account when they are applied on an I/O object. These labels are used to define the I/O observable behavior in KernelCoqa$_{rio}$. Intuitively, the order of GET/SET operations on an I/O object is the behavior KernelCoqa$_{rio}$ manifests itself to the outside world.

Definition 20 (I/O Step)**.** *An* I/O step *is a local step* st_r *with a label* $\textsc{Set}(t, \gamma, v)$ *or* $\textsc{Get}(t, \gamma, v)$ *where* $H(\gamma) = \langle cn, W, R, F\rangle$ *and* $cn =$ IOClass*. We use* st_r^{γ} *to denote that* st_r *is an I/O step accessing the I/O object* γ*.*

Unlike KernelCoqa$_{fio}$, KernelCoqa$_{rio}$'s I/O observable behavior and equivalence is not a superset of the original formalization established in Section 3.3. This is because observable behaviors defined in Definition 4 and Definition 17 are not adequate to capture I/O observable behavior in KernelCoqa$_{rio}$: $ob(p)$ does not distinguish I/O objects from common heap objects so it fails to capture messages sent to I/O objects, those messages are external visible; $ob_{fio}(p)$ is only able to capture certain I/O behavior – ones exhibited on nonlocal steps. So KernelCoqa$_{rio}$ needs to redefine observable behavior.

Definition 21 (I/O Observable Behavior ob^o_{rio})**.** *Given a path* p *and an I/O object* o*, the* I/O observable behavior *of* p *with respect to* o*,* $ob^o_{rio}(p)$*, is the sequence of labels of* $\textsc{Get}(t, o, v)$ *and* $\textsc{Set}(t, o, v)$ *occurring in* p*.*

Definition 22 (I/O Observable Equivalence $\equiv_{rio}$)**.** *Two paths* p_1 *and* p_2 *are I/O observable equivalent, written* $p_1 \equiv_{rio} p_2$*, iff for any I/O object* o*,* $ob^o_{rio}(p_1) = ob^o_{rio}(p_2)$*.*

Definition 23 (I/O Equivalent Step Swap)**.** *For two consecutive steps $st_{r_1} st_{r_2}$ in a computation 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 $p \equiv_{rio} p'$ and $st'_{r_2} st'_{r_1} = (S, r_2, S^*)(S^*, r_1, S'')$, then we say it is an* I/O equivalent step swap.

The following lemma is a strengthened version of the Lemma 2 (Bubble-down Lemma) in the sense that it states the step swap taken in the Bubble-down Lemma also preserves I/O equivalence defined by Definition 22. The proof technique and procedure hence are similar to those of the Bubble-down lemma in Section 3.3.

Lemma 4 (Bubble-down with $\equiv_{rio}$)**.** *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 st_{r_1} is a local step, then a step swap of st_{r_1} and st_{r_2} is an I/O equivalent step swap.*

Proof. First, observe that if t_2 is a subtask of t_1, then it is impossible for st_{r_1} to be a local step while st_{r_2} is a step of t_2. Because according to semantics defined in Figure 3.4 a task and its subtask never have their local steps consecutive to each other, they are demarcated by some nonlocal steps as demonstrated in Figure 3.5.

st_{r_1} is a local step, it can at most change one object's state on the heap H and a local step does not change N according to the operational semantics. So $st_{r_1} = ((H_1, N_1, T_1), r_1, (H_2, N_1, T_2))$ where H_1 and H_2 differ at most on one object o.

(Case I) st_{r_2} is also a local step.

According to Definition 21 and 22, $\equiv_{rio}$ is defined by the label sequence of I/O steps on any given I/O object o occurring in path p. So if st_{r_2} is also a local step, we need to show that st_{r_1} and st_{r_2} cannot be I/O steps on the same I/O object so I/O observable

equivalence still holds with a step swap of $st_{r_1} st_{r_2}$.

Because st_{r_2} is a local step, it does not change N. Let $st_{r_2} = ((H_2, N_1, T_2), r_2, (H_3, N_1, T_3))$. Because st_{r_1} and st_{r_2} are steps of different tasks t_1 and t_2, st_{r_1} and st_{r_2} must change different elements of T (t_1 and t_2 respectively). This means any change made by st_{r_1} and st_{r_2} to T always commute. So in the remaining proof of **Case I**, we focus on changes st_{r_1} and st_{r_2} make on H, and omit N and T for concision.

Let $change(st_{r_1}) = (s_{o_1}, r_1, s'_{o_1})$ and $change(st_{r_2}) = (s_{o_2}, r_2, s'_{o_2})$.

If $o_1 \neq o_2$, then $change(st_{r_1} st_{r_2}) = ((s_{o_1}, s_{o_2}), r_1 r_2, (s'_{o_1}, s'_{o_2}))$. Swapping the order of st_{r_1}, st_{r_2} by applying r_2 first then r_1 results in the change $((s_{o_1}, s_{o_2}), r_2 r_1, (s'_{o_1}, s'_{o_2}))$, which has the same start and end states as $change(st_{r_1} st_{r_2})$, regardless of what r_1 and r_2 are. Moreover, since $o_1 \neq o_2$, if any or both of o_1 and o_2 are I/O objects, the step swap of $st_{r_1} st_{r_2}$ does not change the I/O observable behavior with respect to o_1 or/and o_2. Therefore it is an equivalent step swap.

If $o_1 = o_2 = o$, then $change(st_{r_1}) = (s_o, r_1, s'_o)$, $change(st_{r_2}) = (s'_o, r_2, s''_o)$ and $change(st_{r_1} st_{r_2}) = (s_o, r_1 r_2, s''_o)$. Let $s_o = \langle cn; R; W; F \rangle$. By inspection of the rules this case only arises if both of the rules are amongst Inst, Get and Set.

Subcase a: $r_1 =$ Inst.

$change(st_{r_1}) = (\mathbf{null}, r_1, s_o)$ so r_2 cannot be Inst since o cannot be created twice. And, r_2 also cannot be any other local rule: o is just created in st_{r_1} by t_1. For t_2 to be able to access o, it must obtain a reference to o first. Only t_1 can pass a reference of o to t_2 directly or indirectly. As a result, if st_{r_2} is a local step of t_2 that operates on o, it cannot be consecutive to st_{r_1}, and vice visa. Therefore, $r_1 =$ Inst is not possible.

Subcase b: $r_2 = \textsc{Inst}$. Trivially, r_2 cannot be INST that creates o no matter what r_1 is because no steps can operate on an object before it is created.

Subcase c: o is not an I/O object.

Because o is not an I/O object, a step swap of $st_{r_1} st_{r_2}$ does not change I/O observable behavior defined on I/O objects. Hereby to show that the step swap of $st_{r_1} st_{r_2}$ is an I/O equivalent step swap we only need to show that the step swap is equivalent on H.

Subcase c1: $r_1 = \textsc{Get}$

If the class of o is declared as `exclusive`, then $change(st_{r_1}) = (\langle cn; R; W; F\rangle, r_1, \langle cn; R; W'; F\rangle)$, $W' = W \cup \{t\}$ if t_1 captures o in st_{r_1} or $W' = W$ if W is a subset of the ancestors of t_1. Either way, there does not exist a consecutive st_{r_2} of t_2 where r_2 is either GET or SET because if so, firing up st_{r_2} would violate the preconditions of the GET or the SET rule.

If the class of o is not declared as `exclusive`, then $change(st_{r_1}) = (\langle cn; R; W; F\rangle, r_1, \langle cn; R \cup \{t_1\}; W; F\rangle)$. If $W \neq \emptyset$, then there does not exist a st_{r_2} of t_2 where r_2 is either GET or SET because if so st_{r_2} would violate the precondition of the GET or the SET rule. In the case of $W = \emptyset$, r_2 can only be GET on o because if it is SET, firing up st_{r_2} would violate the precondition of SET. Therefore, $change(st_{r_1} st_{r_2}) = (\langle cn; R; W; F\rangle, r_1 r_2, \langle cn; R \cup \{t_1\} \cup \{t_2\}; W; F\rangle)$. Swapping the application order of r_1 and r_2 we get $change(st'_1 st'_2) = (\langle cn; R; W; F\rangle, r_2 r_1, \langle cn; R \cup \{t_2\} \cup \{t_1\}; W; F\rangle)$, which has the same start and end states as $st_{r_1} st_{r_2}$ because set union commutes.

Subcase c2: $r_1 = \textsc{Set}$.

Let $st_{r_1} = (\langle cn; R; W; F\rangle, r_1, \langle cn; R; W'; F'\rangle)$ and $t_1 \in W'$. Then r_2 cannot be GET

or SET because if so st_{r_2} would violate the precondition of the GET or the SET rule.

Subcase d: o is an I/O object.

In this subcase, we prove that if st_{r_1} is an I/O access on the I/O object o, then there does not exist a valid st_{r_2} such that it also accesses the same object o.

Subcase d1: $r_1 = \text{GET}$.

If o is an I/O object, then there does not exist a valid st_{r_2} for the same reason stated in **Subcase b1** where the class of o is declared as `exclusive`. This means that it is impossible for st_{r_2} to be $st^o_{r_2}$ if st_{r_1} is a GET step on o.

Subcase d2: $r_1 = \text{SET}$.

Let $st_{r_1} = (\langle cn; R; W; F\rangle, r_1, \langle cn; R; W'; F'\rangle)$ and $t_1 \in W'$. Then r_2 cannot be GET or SET because if so st_{r_2} would violate the precondition of the GET or the SET rule. Namely, it is impossible for st_{r_1} and st_{r_2} to be two local steps operate on the same I/O object o.

(Case II) st_{r_2} is a nonlocal step.

Let $st_{r_1} = ((H_1, N_1, T_1), r_1(H_2, N_1, T_2))$ and H_2 differs from H_1 at most on object o since st_{r_1} is a local step. Because st_{r_1} and st_{r_2} are steps of t_1 and t_2 respectively, they must change different elements of T, i.e., t_1 and t_2. st_{r_2} as a nonlocal step may add a new fresh element t to T. But st_{r_1} and st_{r_2} still obviously commute in terms of changes to T. So in the following proof, we omit T for concision.

Because st_{r_2} is assumed to be a nonlocal step, it is impossible to get a different I/O observable behavior from p by a step swap of a local step st_{r_1} and a nonlocal step st_{r_2}. Therefore, to show that the step swap of $st_{r_1}st_{r_2}$ is I/O equivalent we only need to show that if $st_{r_1}st_{r_2} = (S, r_1, S')(S', r_2, S'')$, then the step swap of $st_{r_1}st_{r_2}$ is $st'_{r_2}st'_{r_1} =$

$(S, r_2, S^*)(S^*, r_1, S'')$.

Subcase a: $r_2 = \textsc{Task}$.

Let t be the new task created in st_{r_2}, then $st_{r_2} = ((H_2, N_1, T_2), r_2, (H_2, N_1, T_2 \cup t))$. The final state change of taking st_{r_1} and st_{r_2} in this order is $((H_1, N_1, T_1), r_1 r_2, (H_2, N_1, T_2 \cup t))$. Swapping st_{r_1} and st_{r_2} results consecutive steps $st'_1 = ((H_1, N_1, T_1), r_2, (H_1, N_1, T_1 \cup t))$ and $st'_2 = ((H_1, N_1, T_1 \cup t), r_1, (H_2, N_1, T_2 \cup t))$, which makes the combined state change of $st'_1 st'_2$ to be $((H_1, N_1, T_1), r_2 r_1, (H_2, N_1, T_2 \cup t))$, the same state change as that of $st_{r_1} st_{r_2}$.

Subcase b: $r_2 = \textsc{SubTask}$.

Let t be the new task created in st_{r_2}, then $st_{r_2} = ((H_2, N_1, T_2), r_2, (H_2, N_1 \cup \{t_2 \mapsto t\}, T_3))$. $st_{r_1} st_{r_2} = ((H_1, N_1, T_1), r_1 r_2, (H_2, N_1 \cup \{t_2 \mapsto t\}, T_3))$. If we apply r_2 first, we get $st'_1 = ((H_1, N_1, T_1), r_2, (H_1, N_1 \cup \{t_2 \mapsto t\}, T_2))$. Then r_1 is applied to get $st'_2 = ((H_1, N_1 \cup \{t_2 \mapsto t\}, T_2), r_1, (H_2, N_1 \cup \{t_2 \mapsto t\}, T_3))$. Therefore, $st'_1 st'_2$ results in a transition $((H_1, N_1, T_1), r_2 r_1, (H_2, N_1 \cup \{t_2 \mapsto t\}, T_3)$ which has the same start and final states as $st_{r_1} st_{r_2}$.

Subcase c: $r_2 = \textsc{End}$.

Let $st_{r_2} = ((H_2, N_1, T_2), r_2, (H_3, N_1, T_3))$, where

$H_3 = \biguplus_{H_2(o) = \langle cn; R; W; F \rangle} (o \mapsto \langle cn; R \backslash t_2; W \backslash t_2; F \rangle)$. Namely, st_{r_2} removes t_2 from R and W sets of all objects in H_2. Consider the case when st_{r_1} changes an object o's F but not its R or W: taking st_{r_1} then st_{r_2} has the same state change as taking the two steps in reversed order because the two steps work on different regions of the heap. If st_{r_1} also changes R or W of o, then it can only add t_1 to R or W of o, while st_{r_2} may only remove t_2 from R or W of o. Consequently, swapping the two steps we still get the same result because the set

operations commute since $t_1 \neq t_2$.

Subcase d: $r_2 = \text{STEND}$.

Let $N(t) \mapsto t_2$, then $st_{r_2} = ((H_2, N_1, T_2), r_2, (H_3, N_1 \backslash t, T_3))$, where

$H_3 = \biguplus_{H_2(o)=\langle cn;R;W;F\rangle} (o \mapsto \langle cn; R\backslash t_2; W\backslash t_2; F\rangle)$. Namely, st_{r_2} removes t_2 from R and W sets of all objects in H_2. Consider the case when st_{r_1} changes an object o's F but not its R or W: taking st_{r_1} then st_{r_2} has the same state change as taking the two steps in reversed order because the two steps work on different regions of the heap. If st_{r_1} adds t_1 to R or W of o, swapping the two steps we still get the same result because the set operations commute since $t_1 \neq t_2$.

□

Theorem 5 (Quantized Atomicity with $\equiv_{rio}$)**.** *For all paths p there exists a quantized path p' such that $p' \equiv_{rio} p$.*

Proof. The proof is similar to that of Theorem 1 but uses Lemma 4 on each induction step to turn p_i to p_{i+1}. □

4.3 Getting Bigger Atomic Regions

Stronger atomicity property can be achieved if quanta of a task which may spread through a computation path can be joined as consecutive *pmsp*'s. In this section, we approach this goal via the following steps: Theorem 5 guarantees the existence of a quantized I/O observably equivalent path for any computation path. This allows us to focus our discussion on quantized paths in the rest of formalization. Then, we establish an I/O equivalent quantum swap (Theorem 6) on a quantized path, which allows two disjunctive

quanta of a task to be able to be moved adjacent to each other. Based on this result, we further study what types of quanta can possibly form a compound quantum which is a bigger atomic subpath consisting multiple quanta of a task.

We first define two critical conditions for assuring shuffling *pmsp*'s in a valid way. Creation contention describes a contention between a task t and its child tasks created by t. The creation contention free ensures that any valid computation path should not have steps of a child task occurring before the step that creates it. Capture contention is a contention between two tasks t and t' who compete to access a shared object. If t write captures the object before t' in a given computation path, then the capture order has to be respected in an equivalent computation path.

Definition 24 (Creation Contention Free)**.** *Given a path p with two consecutive steps $st_{r_1} st_{r_2}$ where $st_{r_1} \in pmsp_{t_1}$, $st_{r_2} \in pmsp_{t_2}$, $t_1 \neq t_2$ and t_2 is not a subtask of t_1, if it is not the case that st_{r_1} has a label of* $\textsc{Task}(t_1, \gamma, mn, v, o, t_2)$ *(t_1 creates t_2), then st_{r_1} and st_{r_2} are* creation contention free.

In the above definition, suppose st_{r_1} has a label of $\textsc{Task}(t_1, \gamma, mn, v, o, t')$ or $\textsc{Sub-Task}(t_1, \gamma, mn, v, o, t')$, then st_{r_1} is the creation step of t'.Any step swap that moves a step of t' be in front of st_{r_1} cannot be valid.

Definition 25 (Capture Contention Free)**.** *Given a quantized path with consecutive steps $st_{r_1} st_{r_2}$ where $st_{r_1} \in pmsp_{t_1}$, $st_{r_2} \in pmsp_{t_2}$, $t_1 \neq t_2$, $st_{r_1} = ((H_1, N_1, T_1), r_1, (H_2, N_2, T_2))$, st_{r_1} is a nonlocal step and st_{r_2} is a local step and change$(st_{r_2}) = (s_o, r_2, s'_o)$, then st_{r_1} and st_{r_2} are* capture contention free *if $H_1\{o\} = \langle cn; R, W; F\rangle$ and $t_1 \notin W$.*

Capture contention free definition means that a local step of t_2 cannot operate on

an object o that has already been write captured by t_1, until t_1 frees o upon its finish.

Definition 26 (CCC-free)**.** *Given a quantized path with consecutive steps $st_{r_1} st_{r_2}$, $st_{r_1} st_{r_2}$ are* CCC-free *iff they are creation contention free and capture contention free.*

Now, we introduce the core Theorem *I/O Equivalent Quantum Swap*. Figure 4.5 shows the procedure and framework of proving the theorem. First, we prove Lemma 5, a lemma that allows a nonlocal step of a task to be swapped with the adjacent local step of another task if the two steps are creation and capture contention free. Figure 4.5 shows that by repeated application of Lemma 5, st_{r_1} can be moved to be in front of st_{r_2} to give a new path p_1 such that $p \equiv_{rio} p_1$. Then, Lemma 6 is introduced in which a step swap of two creation contention free nonlocal steps gives a new path that is $\equiv_{rio}$ to the original path. Two nonlocal steps st_{r_1} and st_{r_2} in p_1 are swapped to give p_2 by Lemma 6 such that $p_1 \equiv_{rio} p_2$. It is easy to see that p_2 is not a quantized path any more. However, we have already proved in Theorem 5 that for any given path we can always find a quantized path that is I/O equivalent to the original path. Consequently, a quantized p' can be obtained via Theorem 5, as shown in Figure 4.5. By transitivity of equivalence, $p \equiv_{rio} p_1 \equiv_{rio} p_2 \equiv_{rio} p'$, so $p' \equiv_{rio} p$. And by then, the I/O Equivalent Quantum Swap Theorem (Theorem 6) can be established. Essentially, Theorem 6 states that two consecutive *pmsp*'s can be swapped to get an I/O equivalent quantized path under the conditions of CCC-free.

Lemma 5 (Nonlocal-Local Bubble Down)**.** *Given any a quantized path with 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 $t_1 \neq t_2$, st_{r_1} is a nonlocal step and st_{r_2} is a local step, if st_{r_1} and st_{r_2} are CCC-free, then a step swap of $st_{r_1} st_{r_2}$ is an I/O equivalent step swap.*

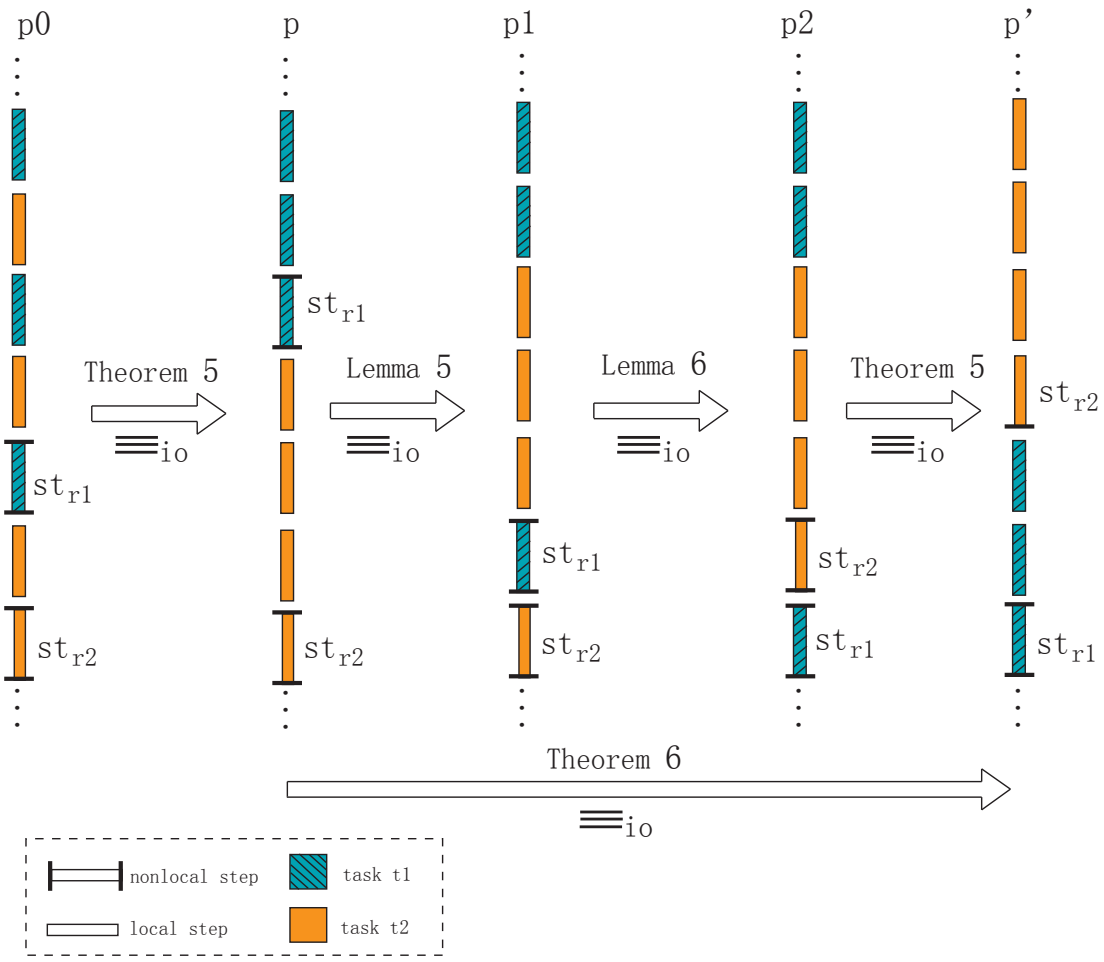

Figure 4.5: Quantized Atomicity in KernelCoqa$_{rio}$

Proof. First, observe that st_{r_1} is a nonlocal step so it does not have an I/O label, while st_{r_2} as a local step may have an I/O label. A step swap of $st_{r_1} st_{r_2}$ does not change the observable behavior of the original path, however, for the step swap of $st_{r_1} st_{r_2}$ to be an I/O equivalent step swap, we need to show that the two steps commute on H, N and T.

Because st_{r_1} and st_{r_2} are steps of different tasks t_1 and t_2, st_{r_1} and st_{r_2} must change different elements of T (t_1 and t_2 respectively) by inspection of the rules. So st_{r_1} and st_{r_2} obviously commute in terms of changes to t_1 and t_2 in T.

(Case I) $r_1 = \textsc{Task}$.

st_{r_2} is a local step so it does not change N, but may only change H. Since st_{r_1} does not change either N or H. st_{r_1} and st_{r_2} obviously commute on N and H.

st_{r_2} also adds a new fresh element t' to T. Due to creation contention free, $t_2 \neq t'$, st_{r_1} and st_{r_2} obviously commute on T because operations on set T commute.

(Case II) $r_1 = \textsc{SubTask}$.

st_{r_2} is a local step so it may change H. st_{r_1} does not change H, so st_{r_1} and st_{r_2} obviously commute on H.

st_{r_1} also adds a new fresh element t' to T and new mapping of $\{t \mapsto t'\}$ to N. Due to creation contention free, $t' \neq t_2$. Moreover, st_{r_2} does not change N and only changes t_2 of T. st_{r_1} and st_{r_2} obviously commute because set operations on set T and N commute.

(Case III) $r_1 = \textsc{TEnd}$.

Let $change(st_{r_2}) = (s_o, r_2, s'_o)$. Because st_{r_1} and st_{r_2} are capture contention free, they work on different objects on the heap so st_{r_1} and st_{r_2} commute in terms of H. Neither st_{r_1} nor st_{r_2} changes N so st_{r_1} and st_{r_2} also commute on N. st_{r_1} and st_{r_2} change different

elements of T (t_1 and t_2 respectively) so they commute on T

(Case IV) $r_1 = \text{STEND}$.

Because st_{r_1} and st_{r_2} are capture contention free, they work on different objects on the heap so st_{r_1} and st_{r_2} commute on H. st_{r_1} removes a mapping $\{t_1 \mapsto t'\}$ from N, $t' \neq t_2$ due to creation contention free. Furthermore,st_{r_2} does not change N so st_{r_1} and st_{r_2} also commute on N. st_{r_1} and st_{r_2} commute on T as well.

□

Lemma 6 (Nonlocal-Nonlocal Bubble Down)**.** *Given any a quantized path p with 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 $t_1 \neq t_2$ and if st_{r_1} and st_{r_1} both are nonlocal steps, if st_{r_1} and st_{r_2} are creation contention free, a step swap of $st_{r_1} st_{r_2}$ is an I/O equivalent step swap.*

Proof. First, observe that st_{r_1} and st_{r_2} are both nonlocal steps so they do not have I/O labels. So a step swap of $st_{r_1} st_{r_2}$ does affect the I/O observable behavior. To show that such a step swap is an I/O equivalent step swap, we only need to show that st_{r_1} and st_{r_2} commute on H, N and T.

Let $st_{r_1} = ((H_1, N_1, T_1), r_1, (H_2, N_2, T_2))$ and $st_{r_2} = ((H_2, N_2, T_2), r_2, (H_3, N_3, T_3))$. Because st_{r_1} and st_{r_2} are steps of t_1 and t_2 respectively, they must operate on different elements of T.

(Case I) $r_1 = \text{TASK}$.

According to the operational semantics defined in Figure 3.4, the TASK rule does not change H and N. So st_{r_1} and st_{r_2} obviously commute on each of H and N.

The TASK rule adds a new element t' to T so $st_{r_1} = (H_1, N_1, T_1), r_1, (H_1, N_1, T_1 \cup$

$\{t'\})$, $t' \neq t_2$ because st_{r_1} and st_{r_2} are creation contention free. st_{r_2} may add another new element t'' to $T_1 \cup \{t'\}$ but st_{r_1} and st_{r_2} commute on T due to commutativity of set union operation.

(Case II) $r_1 =$ SUBTASK.

st_{r_1} does not change H, so st_{r_1} and st_{r_2} commute on H no matter st_{r_2} changes H or not.

Let $st_{r_1} = ((H_1, N_1, T_1), r_1, (H_1, N_1 \cup \{t_1 \mapsto t'\}, T_1 \cup \{t'\}))$, t' is the new subtask created by st_{r_1}. $t' \neq t_2$ because st_{r_1} and st_{r_2} are creation contention free.

Subcase a: $r_2=$ TASK.

Let t'' be the new task created by st_{r_2}, then $st_{r_2} = ((H_1, N_1 \cup \{t_1 \mapsto t'\}, T_1 \cup \{t'\}), r_2, (H_1, N_1 \cup \{t_1 \mapsto t'\} \cup \{t_2 \mapsto t''\}, T_1 \cup \{t'\} \cup \{t''\}))$. Due to commutativity of set union operation, st_{r_1} and st_{r_2} commute on N and T.

Subcase b:$r_2 =$ SUBTASK.

Let t'' be the new task created by st_{r_2}, then $st_{r_2} = ((H_1, N_1 \cup \{t_1 \mapsto t'\}, T_1 \cup \{t'\}), r_2, (H_1, N_1 \cup \{t_1 \mapsto t'\}, T_1 \cup \{t'\} \cup \{t''\}))$. Due to the commutativity of set union operation, st_{r_1} and st_{r_2} commute on N and T.

Subcase c: $r_2 =$ TEND.

st_{r_2} may change H, but st_{r_1} does not. So they commute on H. st_{r_2} does not change N so st_{r_1} and st_{r_1} commute on N. st_{r_2} changes t_2 of T, a different element that st_{r_1} operates on. Moreover, $t' \neq t_2$ where t' is the new element st_{r_1} adds to T. Hence st_{r_1} and st_{r_2} commute on T as well.

Subcase d:$r_2 =$ STEND.

st_{r_2} takes a mapping $\{t_2 \mapsto t''\}$ from N. Because st_{r_1} and st_{r_2} commute on T and $t_1 \neq t_2$, st_{r_1} and st_{r_2} each removes different element from N. st_{r_2} does not affect T. So st_{r_1} and st_{r_2} commute on H, N and T.

(Case III) $r_1 =$ TEND.

Subcase a: $r_2 =$ TASK.

st_{r_1} and st_{r_2} commute on H because only st_{r_1} affects H. None of the two steps affect N so they commute on commute on N as well. st_{r_1} and st_{r_2} operate on different elements of T and st_{r_2} adds a new element to T. Hence st_{r_1} and st_{r_2} as well commute on T

Subcase b: $r_2 =$ SUBTASK. The proof is similar to **Subcase a**.

Subcase c:$r_2 =$ TEND.

Neither st_{r_1} nor st_{r_2} changes N so they commute on N. st_{r_1} and st_{r_2} also commute on T because they change different elements of T. st_{r_1} removes t_1 from W and R set of all objects on the heap H, while st_{r_2} removes t_2 from the W and R of objects. So st_{r_1} and st_{r_2} also commute on H due to the commutativity of set operations.

Subcase d:$r_2 =$ STEND.

st_{r_1} and st_{r_2} commute on T because they affect different elements of T. st_{r_2} removes a mapping $t_2 \mapsto t'$ from N while st_{r_1} does not change N. st_{r_1} removes t_1 from W and R set of all objects on the heap H, while st_{r_2} removes t_2 from the W and R of objects. So st_{r_1} and st_{r_2} also commute on N and H due to commutativity of set operations.

(Case IV) $r_1 =$ STEND.

Subcase a:$r_2 =$ TASK.

st_{r_1} and st_{r_2} commute on both H and N because only st_{r_1} affects on H and N.

st_{r_1} changes t_1 of T while st_{r_2} changes t_2 of T and adds a new element to T. The two steps obviously commute on T as well.

Subcase b:$r_2 =$ SUBTASK.

The proof is similar to **Subcase a**.

Subcase c:$r_2 =$ TEND.

st_{r_1} and st_{r_2} commute on T because they affect different elements of T. st_{r_1} removes t_1 from W and R set of all objects on the heap H, while st_{r_2} removes t_2 from the W and R of objects. So st_{r_1} and st_{r_2} commute on H due to the commutativity of set operations. st_{r_1} removes a mapping $t_1 \mapsto t'$ from N while st_{r_2} does not change N so st_{r_1} and st_{r_2} commute on N as well.

Subcase d: $r_2 =$ STEND.

st_{r_1} and st_{r_2} commute on T because they affect different elements of T. st_{r_1} removes a mapping $\{t_1 \mapsto t'\}$ from N while st_{r_2} removes $\{t_2 \mapsto t''\}$, $t_1 \neq t_2$, so st_{r_1} and st_{r_2} commute in terms of N because set operations commute. Moreover, st_{r_1} removes t_1 from W and R set of all objects on the heap H, while st_{r_2} removes t_2 from the W and R of objects. So st_{r_1} and st_{r_2} also commute on N and H due to the commutativity of set operations. □

Definition 27 (Sequential Steps)**.** *Two steps st_{r_1} and st_{r_2} in a computation path p are* sequential steps *if st_{r_1} occurs before st_{r_2} in p.*

Two sequential steps are ordered steps in p but they do not have to be consecutive to each other.

Theorem 6 (I/O Equivalent Quantum/*pmsp* Swap)**.** *For any quantized path p with two sequential nonlocal steps st_{r_1} and 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 st_{r_1}*

and st_{r_2} are creation contention free and st_{r_1} is also capture contention free with all local steps of $pmsp_{t_2}$, then $pmsp_{t_1}$ can be moved to be after $pmsp_{t_2}$ by a series of equivalent step swap to get a quantized path p' and $p' \equiv_{rio} p$.

Proof. As demonstrated in Figure 4.5, Theorem 6 directly follows Lemma 5, Lemma 6 and Theorem 5. □

With Theorem 6 certain *pmsp*'s of a task can be shuffled to be consecutive to each other to obtain an I/O equivalent path. The sequence of *pmsp*'s of the same task forms a bigger atomic region, a union of these consecutive quanta. Now, we study how to systematically apply the results we have proved so far to achieve a goal of welding disjoint quanta of a task.

Recall that the nonlocal step at the end of a *pmsp* is called the point of the *pmsp*.

Definition 28 (Join Point)**.** *Given a quantized path p with $pmsp_{t,i}$ and $pmsp_{t,i+1}$, if $pmsp_{t,i}$ and $pmsp_{t,i+1}$ can be moved to be adjacent to each other by a series of I/O equivalent step swaps, then the point of $pmsp_{t,i}$ is a* join point*.*

Definition 29 (Compound Quantum)**.** *Given a quantized path p with a sequence of $pmsp_t$'s, $pmsp_{t,i} \dots pmsp_{t,j}$, if there is no pmsp's of other tasks occurring between $pmsp_{t,i}$ and $pmsp_{t,j}$, then $pmsp_{t,i} \dots pmsp_{t,j}$ form a* compound quantum*. We use cp_t to denote a compound quantum of t.*

If $pmsp_{t,i} \dots pmsp_{t,j}$ where $j > i$ form a compound quantum, points of $pmsp_{t,i} \dots pmsp_{t,j-1}$ are all join points. A compound quantum has the same atomicity property as a quantum because it does not interleave with quanta of any other tasks. By the above definition, a compound quantum is a subpath includes more than one *pmsp*'s of a task.

In KernelCoqa$_{rio}$, there are four types of nonlocal steps (TASK, SUBTASK, TEND and STEND) that could be a *pmsp* point. Among the four types of nonlocal step, a TEND/STEND step obviously cannot be a join point because there is no more *pmsp*'s of the same task afterwards. So, in the rest of this section, we focus on the SUBTASK and the TASK steps.

Fact 3 (A SUBTASK Step is Never a Join Point)**.** *Give a quantized path p with $pmsp_{t,i}$ and $pmsp_{t,i+1}$, $pmsp_{t,i}$ has st_{r_1} as its point and $pmsp_{t,i+1}$ has st_{r_2} with its point and $r_1 =$* SUBTASK*, then st_{r_1} is never a join step.*

According to operational semantics defined in Figure 3.4, when t creates a subtask t', it pauses its computation until the subtask t' finishes. Therefore, there is at least one $pmsp'_t$ between $pmsp_{t,i}$ and $pmsp_{t,i+1}$. Suppose the point of $pmsp'_t$ is $st'_{r'}$, then $st'_{r'}$ and st_{r_2} are not creation contention free so that Lemma 6 can not apply. As a result, there does not exist p' such that $p' \equiv_{rio} p$ and $pmsp_{t,i}$ and $pmsp_{t,i+1}$ are adjacent to each other in p'.

Corollary 1 (A TASK Step Can Be a Join Point)**.** *Give a quantized path p with $pmsp_{t,i}$ and $pmsp_{t,i+1}$, with $pmsp_{t,i}$ has st_{r_1} as its point and $pmsp_{t,i+1}$ has st_{r_2} as its point and $r_1 =$* TASK*, if the nonlocal steps between $pmsp_{t,i}$ and $pmsp_{t,i+1}$ are capture contention free with the local steps of $pmsp_{t,i+1}$, then st_{r_1} is a join point.*

Proof. We use *ps* to denote the *pmsp* sequence between $pmsp_{t,i}$ and $pmsp_{t,i+1}$ inclusive. Suppose there are n *pmsp*'s of other tasks between $pmsp_{t,i}$ and $pmsp_{t,i+1}$. We index those *pmsp*'s with their distance with $pmsp_{t,i+1}$ so that $ps = pmsp_{t,i} pmsp^n_{t_k} \dots pmsp^1_{t_1} pmsp_{t,i+1}$, $k \leq n$. We also assume that $pmsp^j_{t_j}$, $j = 1 \dots n$, has a point st'_{r_j}.

Start with $pmsp^1_{t_1}$, st'_{r_1}, the point of $pmsp^1_{t_1}$, and all steps in $pmsp_{t,i+1}$ are creation contention free because if otherwise, it is impossible for steps of t, i.e. $pmsp_{t,i}$, to occur before st'_{r_1} in p. Since st'_{r_1} is also capture contention free with all local steps of $pmsp_{t,i+1}$, by Theorem 6 we can get a path $p_1 \equiv_{rio} p$ and p_1 is same as p except p_1 has ps replaced with $pmsp_{t,i}pmsp^n_{t_k} \ldots pmsp^2_{t_j}pmsp_{t,i+1}pmsp^1_{t_1}$.

By repeated application of Theorem 6 to all $pmsp$'s between $pmsp_{t,i}$ and $pmsp_{t,i+1}$ by the order of their indices, we can eventually get a path p_n such that $p_n \equiv_{rio} p$ and p_n is the same as p except that ps is replaced with $pmsp_{t,i}pmsp_{t,i+1}pmsp^n_{t_k} \ldots pmsp^1_{t_1}$. Therefore st_{r_1} is a join point. □

A join point connects two $pmsp_t$'s to form a compound quantum. And a compound quantum is a sequence of $pmsp_{t,i}$'s, $i = 1 \ldots n$, $n > 2$ where the point of $pmsp_{t,k}$, $k = i \ldots n-1$ is a joint point.

Definition 30 (Maximal Compound Quantum)**.** *Given a quantized path p with cp_t, if there does not exist a $pmsp_t$ either before or after cp_t in p that can be found adjacent to cp_t in a path p' that $p \equiv_{rio} p'$, then cp_t is a* maximal compound quantum*, written as mcp_t.*

Figure 4.6 demonstrates the step-by-step procedure for building up a compound quantum with growing size until a maximal compound quantum reached. In the figure, every *pmsp* is represented by a big step consisting of a representative local step and a nonlocal step, a TASK nonlocal step is labeled with "*" and $p_1 \equiv_{rio} p_2 \equiv_{rio} p3 \equiv_{rio} p4$. One application of Theorem 6 takes p_1 to p_2, with $pmsp_{t,i}$ and $pmsp_{t,i+1}$ closer to each other. The second application of Theorem 6 brings p_2 to p_3 in which $pmsp_{t,i}$ and $pmsp_{t,i+1}$ adjacent to each other to form a compound quantum framed by a dashed box. Corollary 1

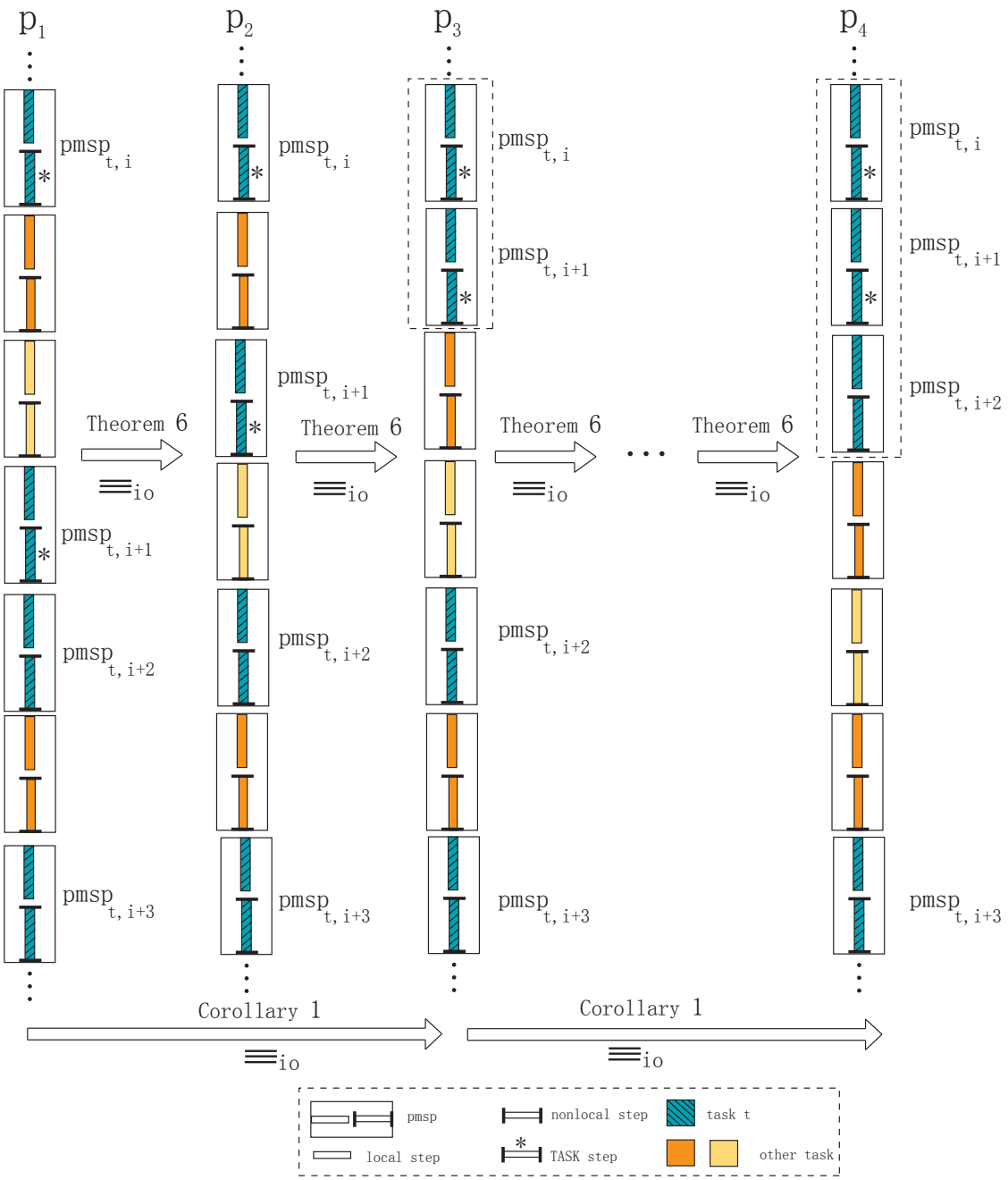

Figure 4.6: Compound Units

is the corollary established on top of Theorem 6, so it takes p_3 to p_4 with $pmsp_{t,i+2}$ being adjacent to $pmsp_{t,i+1}$ which results in an even bigger compound quantum consisting of $pmsp_{t,i}$, $pmsp_{t,i+1}$ and $pmsp_{t,i+2}$. Notice that we are not able to apply either Theorem 6 or Corollary 1 to $pmsp_{t,i+3}$ because the point of $pmsp_{t,i+2}$ is not a TASK step. If $i = 1$, then the compound quantum consisting of $pmsp_{t,1}$, $pmsp_{t,2}$ and $pmsp_{t,3}$ is a maximal compound quantum there does not exist another $pmsp_t$ either before or after it can be shuffled to form an even bigger compound quantum.

The theorem and facts illustrated in this section show that a TASK step is the only type of nonlocal message that could possibly be a join point where two quanta can be welded to form a bigger compound quantum. This fits our intuition illustrated in earlier Section 2.3.4: imaging a system with two tasks t_1 and t_2, t_1 creates t_2 by an asynchronous message. After the creation point, the two tasks run independently without sharing any objects including I/O objects. Then any computation path in the system can be viewed as t_1 runs before t_2, which means the t_1 as a whole is atomic: it has one compound quantum consisting of two quanta demarcated by a TASK step.

4.4 Atomic Tasks

Tasks with all its quanta in a consecutive sequence are atomic in their totality. Not all tasks fall in to this category. For instance, tasks with subtasks obviously are not atomic tasks because the exact purpose of subtasking is to introduce interleavings. However, atomic tasks are an interesting topic to explore. First of all, atomic tasks are tasks with the strongest atomicity property. Secondly, by studying atomic tasks, we understand more

about the similarity and difference between the Coqa model and the Actor model: Coqa subsumes the Actor model in the sense that the Actor model can be encoded in Coqa, while Coqa is stronger than the Actor model since the atomicity in Coqais deep while the Actor model's atomicity is shallow.

Definition 31 (Creation *pmsp* of Task t)**.** *Given a* $pmsp_{t'}$ *with a* Task *step as its point, if the* Task *step has a label* $(t', \gamma, mn, v, o, t)$*, then the* $pmsp_{t'}$ *is the* creation *pmsp* of t*, written as* $pmsp^t_{t'}$*.*

Fact 4. *Every task* t *has exact one creation pmsp except the bootstrapping task* t_{main} *which dose not have a creation pmsp.*

We can introduce a nonlocal Task step with a label $\textsc{Task}(\epsilon, \epsilon, main, v, \epsilon, t_{main})$ as the pseudo step creating t_{main}. And we can safely make the pseudo step as the first step of every computation path, without bringing any changes to our formalization. Now, the bootstrapping task t_{main} also has its creation *pmsp*, $pmsp^{t_{main}}$, consisting of only the pseudo nonlocal Task step.

Definition 32 (Quantum Unit)**.** *For a given quantized path* p*, a* quantum unit *of a task* t *is a sequence of pmsp's of* t*,* $pmsp_{t,i} \dots pmsp_{t,j}$ *in* p*,* $j - i > 0$*. We use* qu_t *to denote the* quantum unit *of* t*.*

By the above definition, a quantum unit can be either a quantum or a compound quantum of a task.

Definition 33 (Singleton Quantum Unit)**.** *For a given quantized path* p*, the* singleton quantum unit *of a task* t *is the sequence of all the pmsp's of* t*,* $pmsp_{t,1} \dots pmsp_{t,n}$ *in* p*,*

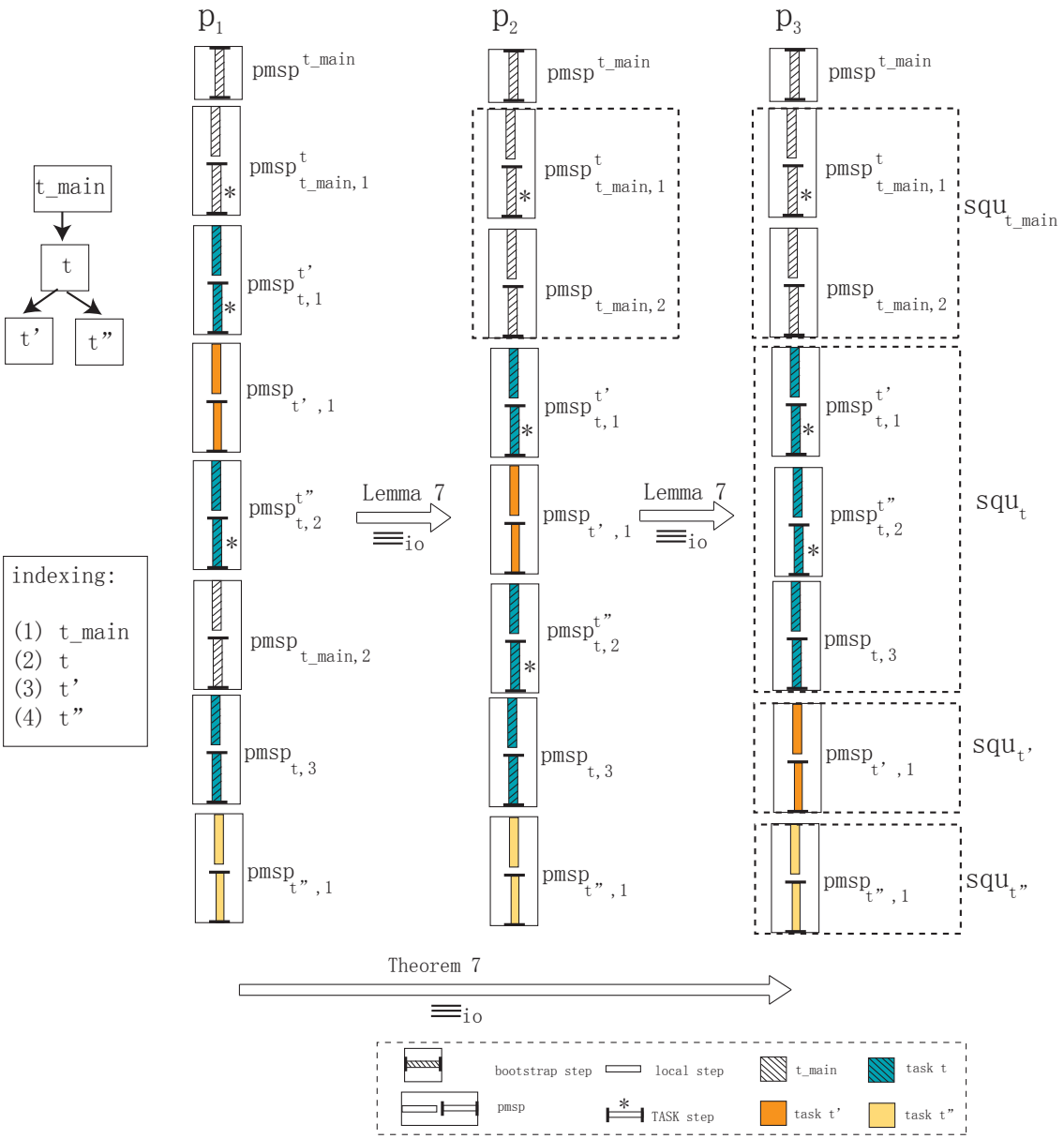

Figure 4.7: Atomic Tasks

$n > 1$ *and there does not exist a* $pmsp_{t,n+1}$. *We use* squ_t *to denote the* singleton quantum unit *of* t.

Obviously, a task may have at most one singleton quantum unit and not all tasks have it. For a given task if there exists an equivalent path with its singleton quantum unit, then the task is atomic as a whole.

Lemma 7. *Given a task* t *in a quantized path* p, *if* t *has no* SubTask *step and for any consecutive steps* st_r *and* $st_{r'}$ *where* $st_r \in pmsp_t$, $st_{r'} \in pmsp_{t'}$, $t \neq t'$, st_r *and* $st_{r'}$ *are capture contention free, then there exists a quantized path* p' *with the* squ_t *and* $p' \equiv_{rio} p$.

Proof. Suppose that task t has n *pmsp*'s in total in p for some n. We proceed by induction on n to show that for all $i \leq n$, $p \equiv_{rio} p_i$ where p_i has a qu_t^i consisting of $pmsp_{t,k}$, $k = 1 \ldots i$. With this fact, for $i = n$, all $pmsp_t$'s can be shuffled to form the squ_t, proving the result.

Let $pmsp_{t'}^t$ be the creation *pmsp* of t. The base case $n = 1$ is trivial since t has only one *pmsp* which is the squ_t. Assumed by induction that we can find a path p_i such that $p \equiv_{rio} p_i$ and p_i has one qu_t^i consisting of $pmsp_{t,k}$, $k = 1 \ldots i$. Notice that qu_t^i must occur after $pmsp_{t'}^t$, the creation *pmsp* of t, in path p_i. Otherwise, p_i is not a valid computation path because no steps in qu_t^i can be taken before t has been created. Due to this fact, steps of other tasks between qu_t^i and $pmsp_{t,i+1}$ can be moved to be after $pmsp_{t,i+1}$ in path p_{i+1} by application of Corollary 1 such that $p_{i+1} \equiv_{rio} p_i \equiv_{rio} p$ and qu_t^{i+1} is formed by the sequence of $pmsp_{t,k}$, $k = 1 \ldots i+1$ in p_{i+1}. Therefore, when $i = n$, a new path p_n can be defined such that $p_n \equiv_{rio} p$ and qu_t^n consists of $pmsp_{t,k}$, $k = 1 \ldots n$. qu_t^n is the squ_t. □

Theorem 7 (Atomic Tasks)**.** *Given a quantized path* p, *if there is no* SubTask *step and for any consecutive steps* st_{r_1} *and* st_{r_2} *where* $st_{r_1} \in pmsp_{t_1}$, $st_{r_2} \in pmsp_{t_2}$, $t_1 \neq t_2$, st_{r_1} *and*

st_{r_2} are capture contention free, then there exists a quantized p' such that $p' \equiv_{rio} p$ and p' consists exact one maximal quantum unit for every task.

Proof. Proceed by first indexing all tasks into a well ordering induced by the ordering of their creation *pmsp*'s in p. Write t_i as the i-th task in this ordering. Suppose that there are n tasks in total in p for some n and $t_1 = t_{main}$ if $n > 0$.

We proceed by induction on n to show for all $i \leq n$, $p \equiv_{rio} p_i$ where t_1 to t_i in this ordering have been shuffled to have exact one squ_{t_i} in a prefix of p_i: $p \equiv p_i = squ_{t_1} squ_{t_2} \dots squ_{t_{i-1}} squ_{(t_i)} \dots$. With this fact, for $i = n$ we have $p \equiv p_n = squ_{t_1} \dots squ_{t_n}$ proving the result.

The base case $n = 0$ and $n = 1$ are trivial. When $n = 0$, the path is empty. When $n = 1$, there is only one task t_{main} in the path and all *pmsp*'s are adjacent to each other forming the $squ_{t_{main}}(squ_{t_1})$.

Assume by induction that t_i for $i < n$ path $p_i = squ_{t_1} \dots squ_{t_i} \dots$ such that $p_i \equiv_{rio} p$. Let $pmsp_{t'}^{t_{i+1}}$ to be the creation *pmsp* of t_{i+1}, then $t' = t_k$, $k \in \{0, \dots, i\}$, otherwise p_i is not a valid computation path because no step of a task can be fired before the task is created. By Lemma 7, *pmsp*s of t_v, $v > i+1$ after squ_{t_i} before the last *pmsp* of t_i on p_i can be moved after the last *pmsp* of t_i, forming a path p_{i+1} such that $p_{i+1} = squ_{t_1} \dots squ_{t_i} squ_{t_{i+1}} \dots$ and $p_{i+1} \equiv_{rio} p_i \equiv_{rio} p$. So when $i = n$, we can find a path p_n such that $p_n = squ_{t_1} \dots squ_{t_n}$. □

Figure 4.7 demonstrates how we prove Lemma 7 and Theorem 7 and their relationship. In the example, there are four tasks in total and the tree on the upper left corner shows their creation relationship: t_{main} creates t which in turn creates t' and t''. p_1 is a complete computation path. First, we index the four tasks according to the order of

their creation *pmsp*'s as shown in the indexing table. Then Lemma 7 is applied to tasks in the indexing order to obtain the singleton quantum unit for each of the tasks, proving Theorem 7.

It is worth mentioning that for a given infinite computation path, Theorem 7 might infinitely delay the execution of a task. Imagine that task t in Figure 4.7 has infinite steps in p_1, then steps of both t' and t'' would be pushed backward to infinite future in p_3. This is a fundamental problem existing in all concurrency formalization systems and hence is also beyond the scope of our model.

Theorem 7 states that all tasks can be atomic if none of them are in a task-subtask relationship and they do not share objects. Technically, even with all the pre-conditions, KernelCoqa$_{rio}$ is still a stronger model than the Actor model because atomic tasks in KernelCoqa$_{rio}$ are bigger atomic units which could include multiple synchronous local messages and involve multiple objects. While the Actor model, the atomic unit is per-message-per-actor. The Actor model can be faithfully implemented in KernelCoqa$_{rio}$, for instance, if actors are realized as objects in KernelCoqa$_{rio}$ and they communicate only using nonlocal asynchronous messages, then a task only includes one message of one actor. So we claim that KernelCoqa$_{rio}$ subsumes the Actor model.

4.5 Further Discussion

The soundness of KernelCoqa$_{rio}$ requires one important assumption about I/O objects: there should no linkage between I/O objects. The two monitors example in Section 2.3.3 is a case of two display objects linked by an observer. The other common form of

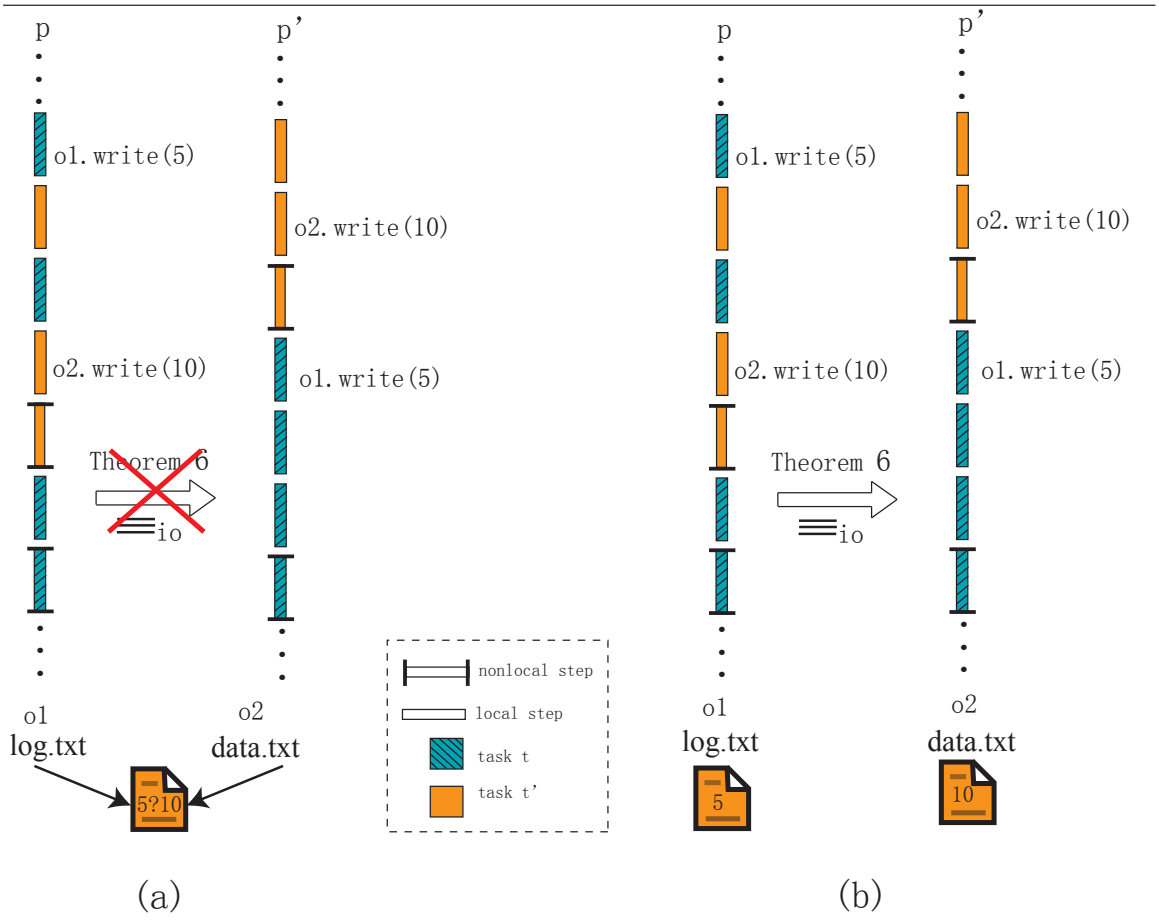

Figure 4.8: Linkage between I/O Objects in KernelCoqa$_{rio}$

linkage is more static, for instance, file aliasing may result in two I/O objects representing the same file on the disk.

The linkage between I/O objects makes them inapplicable in KernelCoqa$_{rio}$ and Figure 4.8(a) explains the reason. The computation path p is I/O equivalent to the quantized path p' according to Theorem 5. However, if both o_1 and o_2 are aliased to "log.txt", then p and p' are not equivalent because p has "10" written in the file while p' would have "5" as the content of the file. Theorem 5 holds only if there is no intrinsic linkage between o_1 and o_2 as shown in Figure 4.8(b).

The linkage between I/O objects is not a provable property in KernelCoqa$_{rio}$, so we have to assume that I/O objects are not linked to each other, which is general the case in practice. Or, if there is any linkage between any I/O objects, it is known to KernelCoqa$_{rio}$ so that linked objects can be treated differently. One simple approach to handle linked I/O objects is to group them into sets such that objects in different sets are not linked. A message sent to any I/O object in such a set would result a write capture of all I/O objects of the set. A wrapper object can also be used to encapsulate linked objects and redirect messages to those objects. But the set approach is more flexible in the sense that individual I/O object in a set is still able to receive messages without redirection. Essentially, KernelCoqa$_{rio}$ manipulates linked I/O objects as if they are one I/O object.

Read-only files are a special case that do not have to follow the restriction that every disk file is represented by one I/O object in KernelCoqa$_{rio}$. Instead, multiple I/O objects of one file can exist for input operations. Each of these read-only I/O objects is able to record its own state (file pointer) so operations performed via it are independent

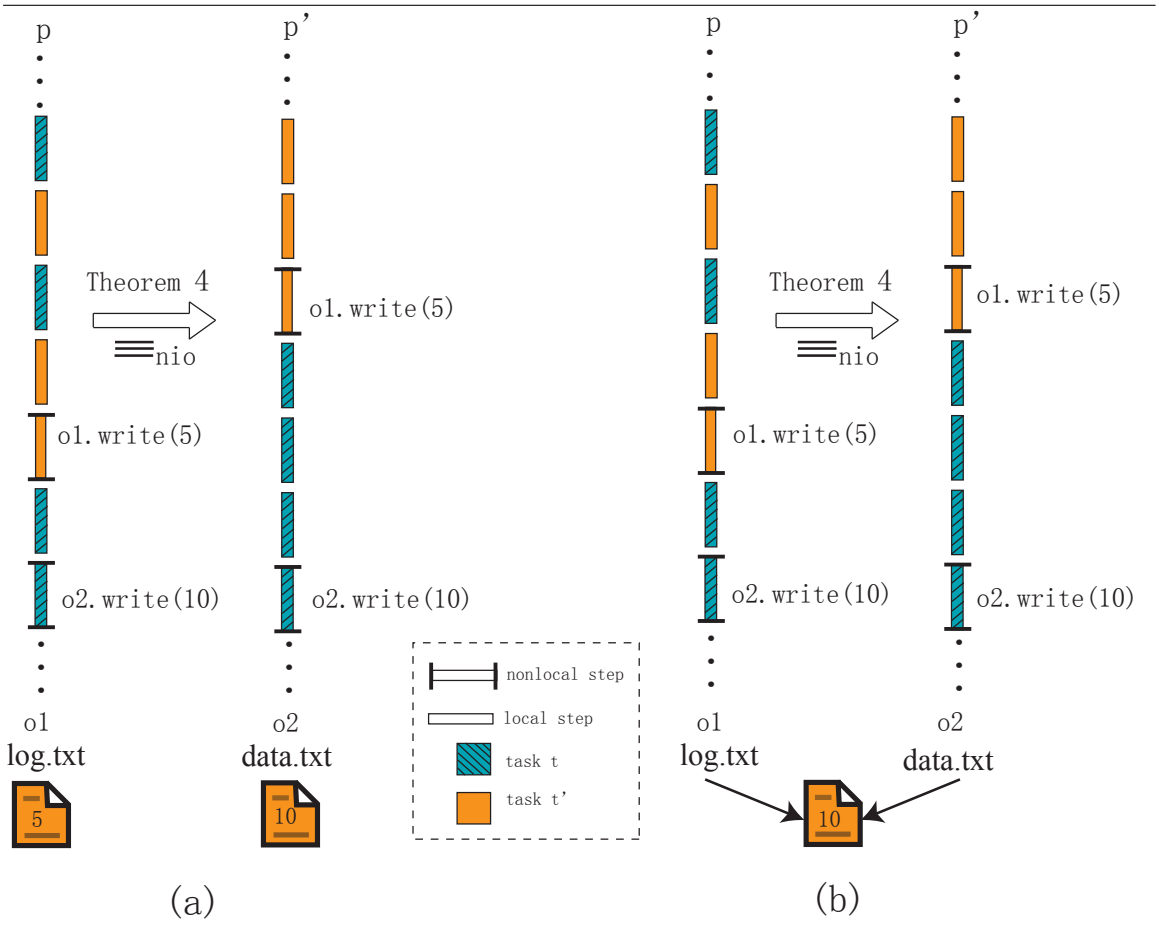

Figure 4.9: Linkage between I/O Objects in KernelCoqa$_{fio}$

from other peers. This can be easily done via adding an instance field to the `IOClass`.

However, the linkage between I/O objects does not affect KernelCoqa$_{fio}$ due to the fact that the order of I/O operations is always preserved regardless of any linkage, as shown Figure 4.9.

As we have mentioned in the beginning of this Chapter, KernelCoqa$_{rio}$ subsumes KernelCoqa$_{fio}$ in expressiveness. An integrated model can be implemented by giving extra type information to I/O objects. More specifically, an I/O object can have a label indicating its type: fair or reserveable. A fair I/O object is restricted to receive only nonlocal messages. Had it receive a local synchronous message, an exception is raised. Such an integrated system can be more flexible in practice and programmers can benefit from advantages of both the fair and the reserveable I/O models.

Chapter 5

CoqaJava: a Coqa Prototype

We have implemented a prototype of Coqa, called CoqaJava, as a Java extension. Polyglot [55] 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. In this chapter, we describe the implementation of CoqaJava and benchmarks we have run on it, then we discuss techniques that may be applied to CoqaJava to further improve its performance.

5.1 CoqaJava Implementation

All language features introduced in Figure 3.1 are included in the prototype. A CoqaJava program is much like a Java program: the `o . m(v)` Java syntax represents a local synchronous message send when used in a CoqaJava program. The main difference is that CoqaJava includes nonlocal synchronous/asynchronous message sending operators `=>` and `->`. CoqaJava also discards Java's threading model in which threads are created and manipulated explicitly via objects of the `java.lang.Thread` class. Instead, parallel

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

Figure 5.1: Translation of the Fragment `bank->transfer()`

```
        class Bank extends CObject {
            ...
            public void transfer(Task t, String from, String to, int b){
                ...
(B1)             Account afrom = (Account)htable.get_del(t, from);
                ...
            }
        }

        class HashTable extends CObject {
            ...
            // Translato-generated delegate method
            Object get_del(Task t, String arg0) {
(B2)            Task subTask = new Task();
(B3)            Task.record(subTask, t);
                Object ret = get(subTask, arg0);
(B4)            subTask.finish();
                return ret;
            }
        }
```

Figure 5.2: Translation of the Fragment `htable ⇒ get()`

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 a need for higher-level coordination and that is planned as future work in Chapter 6.

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 subclasses of this base class.

`CObject` implements the algorithm for checking and updating an object's capture sets, which is activated via invoking the `readReq`/`writeReq` method 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. Figure 5.1 shows by way of example how the translator maps an asynchronous message send at `bank->transfer()` of the earlier example in Figure 2.1: it is wrapped up as an instantiation of a translator-generated inner class `TransferWrap` (line `A1`), in which a new `Task` object is created (line `A3`). Similarly, Figure 5.2 gives an example of the translation for a synchronous message

send at `htable⇒get()` of the earlier example in Figure 2.6: 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 a (sub)task finishes, a call to the method `finish` is invoked on its representative `Task` object (Figure 5.1 line `A4` and Figure 5.2 line `B4`). The finishing process releases all objects captured by the ending task.

We utilize the new `java.util.concurrent` package in our prototype implementation. Every CoqaJava application has a built-in `ScheduledThreadPoolExecutor`, referenced by `rootExecutor`. It serves as an execution engine for the tasks of the application. Figure 5.1 line `A2` shows how a task is submitted to the executor for execution. Subtasks can always be run in the parent thread in CoqaJava because their computations do not overlap.

Figure 5.3 gives the complete code translation schema describing how CoqaJava programs are mapped to Java. The remaining Java ones, including the primitive datatypes, **public** and **private** modifiers for methods *etc.*, that are not shown in this table are generally kept unchanged. We currently leave some other Java language features out of our implementation, including inner classes, native methods, field uses which are not via accessors, exceptions, packages for isolated namespaces, and static fields; these features are not difficult to include. And we will discuss how they can be implemented in CoqaJava in Section 6.3.

As shown in Figure 5.3, every method is translated to carry an extra parameter, `Task t`, that records which task the invocation is in. In CoqaJava, objects are restricted

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

e -> $mn(e')$	`Task t =` **new** Mn`Wrap();`
	`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 => $mn(e')$	$e.mn\_\texttt{del}(\texttt{t}, e')$
	
	`//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 5.3: CoqaJava to Java Syntax-Directed Translation

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 the `readReq`/`writeReq` method 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 the `readReq` method provided by the `CObject` on the current object. If the read request is granted, the `readReq` method 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.

The CoqaJava compiler only translates user defined classes and leaves library classes unchanged. Therefore, a user defined class cannot extend a library class because if it does, then it will not be translated to a subclass of `CObject`. This is a limitation of the prototype implementation but not the language model. A full-fledged implementation can easily extend the prototype to have all classes including library classes to be subclasses of the `CObject` as the way that all classes in Java are subclasses of `java.lang.Object`.

Constructors For simplicity, operational semantics of KernelCoqa does not formalize constructors: they are treated no different from methods. However, we include constructors in CoqaJava for ease of programming. A task does not capture the object it creates although it may initialize fields of the object. This implementation choice is made because a task

exclusively holds the initial reference of an object it creates anyway, so there is no need to for the task to capture the object.

5.2 Benchmark Evaluation

CoqaJava preserves quantized atomicity by default in its language model by using a pessimistic locking scheme implicitly. While it saves programmers from worrying about how to correctly synchronize shared resource accesses themselves, the approach inevitably introduces undesirable impact on the performance of the programs written in Coqa. This motivates us to benchmark CoqaJava. Understanding the performance of CoqaJava helps us to understand the behavior patterns of concurrent programming and tests the practicality of the Coqa language itself.

Although there are no standard benchmarks available for evaluating concurrent languages like CoqaJava on all possible concurrent algorithms or patterns, we can identify two common tasks of concurrent applications and evaluate CoqaJava on the two. The two tasks are summarized well in [27] and quoted as follows:

> Concurrent applications tend to interleave two very different sorts of work: accessing shared data, such as fetching the next task from a shared work queue, and thread-local computation (executing the task, assuming the task itself does not access shared data). Depending on the relative proportions of the two types of work, the application will experience different levels of contention and exhibit different performance and scaling behaviors.
>
> If N threads are fetching tasks from a shared work queue and executing them, and the tasks are compute-intensive and long-running (and do not access shared data very much), there will be almost no contention; throughput is dominated by the availability of CPU resources. On the other hand, if the tasks are very short-lived, there will be a lot of contention for the work queue and throughput is dominated by the cost of synchronization.

We used three benchmark programs. The first program is contention-intensive. It

uses a group of short-lived tasks contending for shared resources. The second and third benchmarks are programs with a set of computation-intensive tasks coordinating with each other. The third benchmark is different from the second one in that it involves tens of thousands of objects in its computation.

The three programs are written in Java, then we modify them in a minimal way to transfer them into CoqaJava benchmark programs. All the benchmarks were performed on a Sun Blade 50 running Solaris 2.5. The relative performance of CoqaJava compared to Java is what we care about in the evaluation. Each benchmark program was run with 7 different thread configurations in which the total number of threads equals to 1, 2, 4, 8, 16, 32 and 64 respectively. With each configuration, the benchmarks of Java version and CoqaJava version were each run 10 times and the average value of the 10 runs was used.

5.2.1 Benchmarks and Results

Sliding Block. Because there are no ready-to-use benchmark programs available for simulating a set of short-lived tasks contending for shared resources as we require, we decided to implement a benchmark program from scratch for the evaluation and we chose the classic "sliding block" puzzle solver problem. The puzzle game starts with blocks of various sizes on a board sitting at some initial positions. Then players need to move those blocks to reach a goal configuration in which blocks all sit at their predefined final positions. Blocks can only be moved one at a time and one step at a time. Every movement has to be on the plane of the board.

We used a brute force algorithm in implementing the puzzle solver problem – checking all possible configurations in a breath-first search fashion until the goal configuration is

Number of Threads	CoqaJava		Java		Ratio
	Mean(seconds)	std	Mean(seconds)	std	
1	0.573	0.001	0.189	0.001	3.030
2	0.532	0.007	0.193	0.005	2.754
4	0.473	0.004	0.200	0.004	2.359
8	0.482	0.005	0.199	0.008	2.417
16	0.504	0.008	0.197	0.008	2.557
32	0.576	0.062	0.237	0.014	2.424
64	0.547	0.030	0.241	0.032	2.265

Table 5.1: Java and CoqaJava on the Puzzle Solver Benchmarks

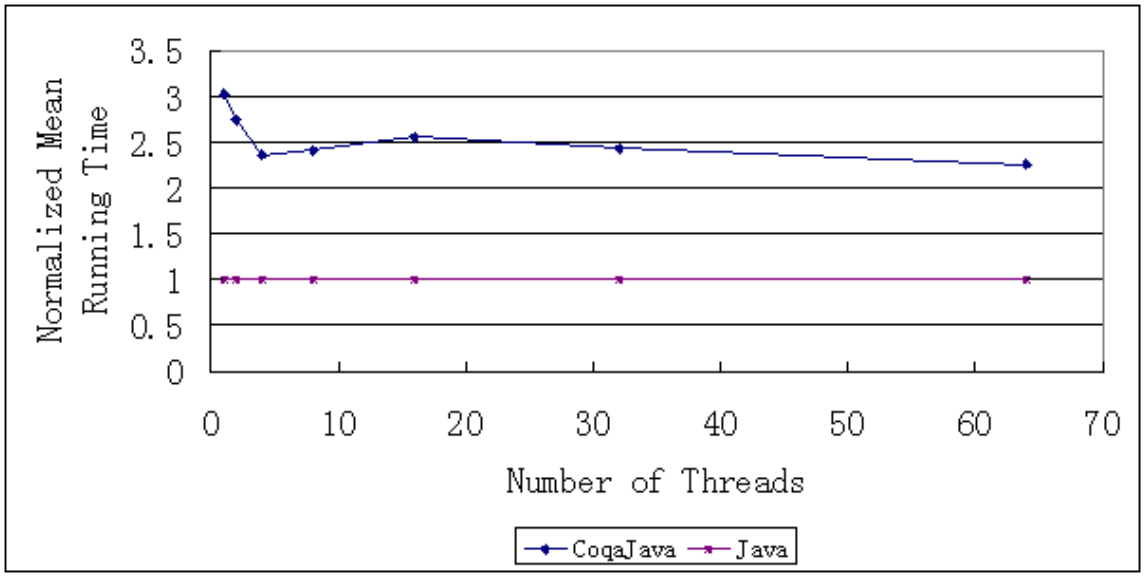

Figure 5.4: Java and CoqaJava on the Puzzle Solver Benchmarks

reached. The producer-consumer pattern was used in the program where tasks constantly take a configuration from a shared queue, check if the configuration is the final goal and put back to the queue all possible moves from the configuration if it is not the goal yet. The puzzle solver game was implemented in Java and CoqaJava with minimal difference in code. The benchmark performance was measured in time that the solver program spent until it found the goal configuration. The benchmark programs in Java and CoqaJava are always run with the same initial and final configuration. Table. 5.1 shows the mean values of the time measurements of Java version and CoqaJava version benchmarks in seconds. The "std" columns are the standard deviations from the mean values. The last column of the table lists the ratio of the mean values of the CoqaJava and Java versions.

As we can see from Table 5.1 and Figure 5.4, the CoqaJava benchmark is up to three times slower than the Java running times. The overhead comes from the fact that CoqaJava performs implicit object capture to ensure quantized atomicity. Specifically, in CoqaJava, if any of the fields of any object is accessed, a read or write locking is performed on the object. There is obviously more locking and unlocking going on in CoqaJava than in Java in which no lock is used unless programmers explicitly code so. Other factors such as the length of calling chain and the duration of an object being captured matter tremendously as well. For instance, locking an object twice is much more expensive than locking it once for two accesses. If those parts of overhead which both CoqaJava and Java benchmarks have to pay are outstanding enough, they offset the performance penalty brought by CoqaJava.

Java Grande SOR Benchmark. To evaluate the case of coordinated computation-intensive tasks, we made a CoqaJava version of the SOR benchmark from the Java Grande

Number of threads	CoqaJava		Java		Ratio
	Mean(seconds)	std	Mean(seconds)	std	
1	13.910	0.011	13.812	0.016	1.007
2	14.023	0.015	13.927	0.010	1.007
4	14.194	0.020	14.171	0.015	1.002
8	14.403	0.020	14.384	0.022	1.001
16	14.681	0.015	14.650	0.017	1.002
32	15.036	0.015	14.992	0.020	1.003
64	15.568	0.057	15.564	0.021	1.000

Table 5.2: Java and CoqaJava on the SOR Benchmarks

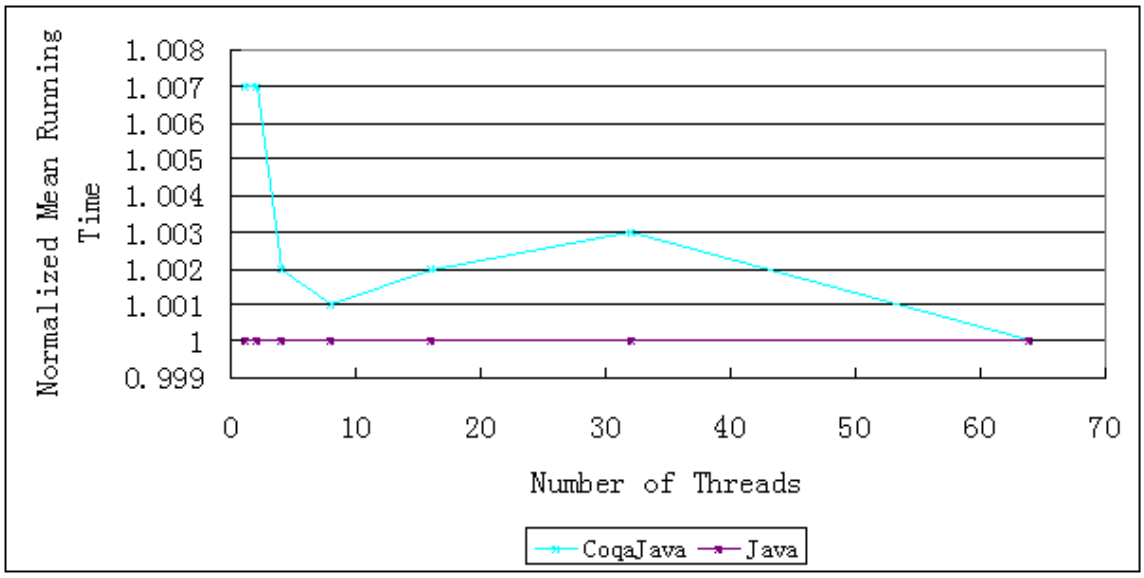

Figure 5.5: Java and CoqaJava on the SOR Benchmarks

Forum Multi-threaded Benchmarks [38]. And compare its performance against the original Java version benchmark. The SOR benchmark performs 100 iterations of successive over-relaxation on a N x N grid (with N = 1000 in our test case), and the algorithm was optimized for parallelization where only nearest neighbor synchronization is required. The mean time measurements in seconds are shown in Table. 5.2 together with the standard deviation of the time measurement for both Java version and CoqaJava version benchmarks. The result shows that CoqaJava version performed comparable to that of original Java version. The extensive computation performed in the benchmark made the overhead of locking scheme introduced in CoqaJava negligible as expected. The last column of the table shows the ratio of the mean time measurements of CoqaJava benchmark against Java benchmark. Figure 5.5 depicts this ratio in a more direct way.

Java Grande Raytracer Benchmark This benchmark measures the performance of a 3D raytracer that involves intensive computations on tens of thousands of objects. The original benchmark operates on a scene containing 64 spheres with a resolution of 150 x 150. We reduced the scene to contain 8 spheres with a resolution of 64 x 32 pixels to lower the memory requirement on our testing machine.

As shown in Table 5.3 and Figure 5.6, the slowdown of CoqaJava is huge, 17.9 times slower than Java on average. The overhead comes from the fact that there are around 60 thousands local objects created in the benchmark execution. Accessing these local objects leads to vast numbers of method invocations on their getter/setter methods and accompanying locking procedure, while the Java version defines almost all the instance object fields as public and accesses them directly. However, because those temporary objects

Number of threads	CoqaJava		Java		Ratio
	Mean(seconds)	std	Mean(seconds)	std	
1	4.907	0.013	0.224	0.018	21.935
2	4.757	0.008	0.235	0.100	20.269
4	4.399	0.017	0.237	0.147	18.553
8	4.330	0.010	0.241	0.069	17.989
16	4.558	0.008	0.253	0.128	18.002
32	4.772	0.012	0.295	0.094	16.165
64	5.025	0.010	0.406	0.264	12.371

Table 5.3: Java and CoqaJava on the Raytracer Benchmarks

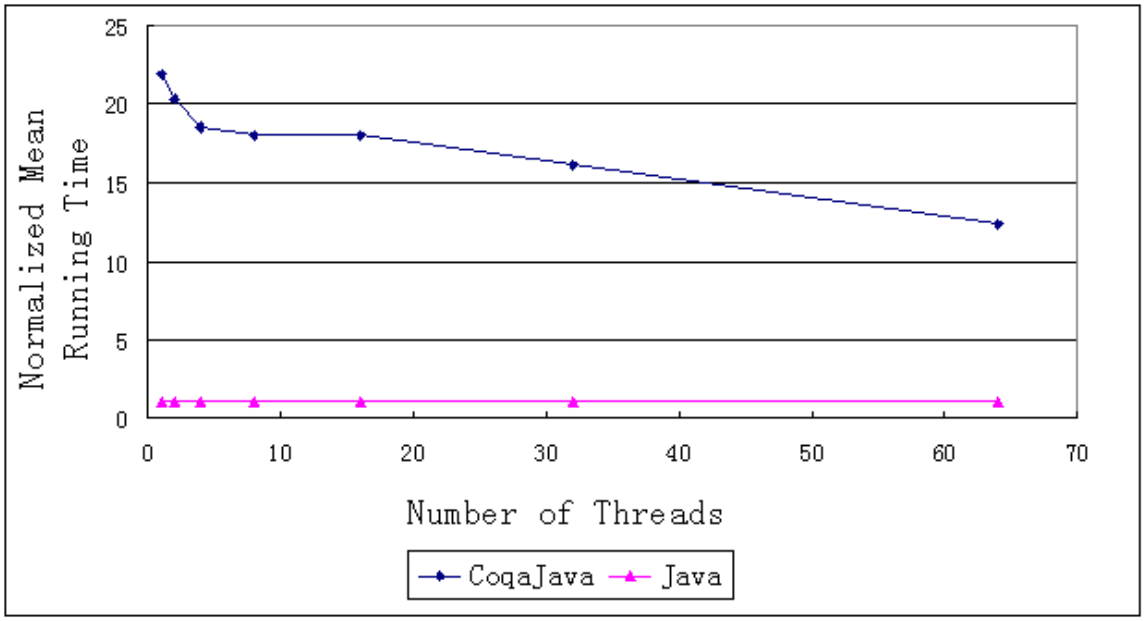

Figure 5.6: Java and CoqaJava on the Raytracer Benchmarks

are all local to their creating tasks and never shared, the locking of those objects are unnecessary. As we will see from the following discussion, the overhead can be greatly reduced with application of some simple optimization techniques.

5.2.2 Optimization

In this section, we optimize CoqaJava with some straightforward strategies and show that those preliminary optimizations can greatly reduce the overhead of CoqaJava.

Number of Threads	CoqaJava		Java		Ratio
	Mean(second)	std	Mean(second)	std	
1	0.374	0.002	0.189	0.001	1.980
2	0.379	0.005	0.193	0.005	1.959
4	0.435	0.007	0.200	0.004	2.171
8	0.430	0.011	0.199	0.008	2.156
16	0.428	0.022	0.197	0.008	2.173
32	0.412	0.017	0.237	0.014	1.736
64	0.472	0.009	0.241	0.032	1.956

Table 5.4: Java and Optimized CoqaJava(1) on the Puzzle Solver Benchmarks

Optimized CoqaJava(1) In the original CoqaJava prototype implementation, each object field access is guarded either by a read lock or by a write lock, which may not be necessary all the time. For example, if a field is defined as **final**, then its value is guaranteed to stay the same, so we can remove the locking procedure on the access of the field without sacrificing the correctness of the program and reduce the overhead in the same time. Another straightforward optimization of the prototype lies in subtask creations. For example, if there is a code snippet in a program as follows:

```
for(i = 0; i < 100 ; i++) {
```

```
        o=>doSomething();
    }
```

then 100 subtasks will be created within the for loop despite the fact that the 100 subtasks are the exact same `Task` objects. We can optimize this behavior by maintaining a ready-to-use `Task` object pool, and re-initialize the object as necessary when a new subtask is required instead of allocating a new `Task` object from heap each time. We applied these two optimization strategies to the CoqaJava and name it optimized CoqaJava(1). Then, we measured the performance of the sliding block puzzle solver again on optimized CoqaJava(1). The result is shown in Table. 5.4 which illustrates the slowdown of the puzzle solver benchmark in optimized CoqaJava(1) is up to 2.2 times, compared to the maximum of 3 times slowdown in CoqaJava.

However, optimized CoqaJava(1) has negligible performance gain on the raytracer benchmark. The obvious reason is that the overhead of raytracer benchmark comes from locking/unlocking massive number of, tens of thousands of, local objects and method invocations on their accessor methods needed for accessing these objects. Such overhead can not be addressed by using the **final** modifier and a `Task` object pool. Fortunately, most of the objects in the raytracer benchmark are created and used by only a single one task so they do not incur any access contention among concurrent threads. Removing locking/unlocking operations on these local task objects is therefore safe and reduces the overhead.

Optimized CoqaJava(2) In order to benchmark a performance improvement over removal of locks on task-local objects, we can add an extra flag to all objects indicating they are task local objects or not. A task can always access its local objects without locking

them. In theory, marking objects as local ones can be successfully done by analyzing CoqaJava programs statically. For instance, there are static analysis tools such as [60] that explore the shared/unshared objects among multiple threads. Unfortunately, we did not find off-the-shelf versions of such static analysis tools to apply to CoqaJava directly. So we decided to emulate such an analysis manually: getting rid of locking operations on obvious task-local objects by hand. And we conducted these manual changes on top of the CoqaJava(1) and call it optimized CoqaJava(2). We believe benchmarking optimized CoqaJava(2) would help us to understand the performance overhead of CoqaJava deeply. The manual annotation we performed is straightforward. We anticipate it can be easily replaced by an automatic static annotation later and we leave it as our future work because it is not technically difficult.

Table 5.5 lists the benchmark results of the sliding block puzzle solver benchmark in optimized CoqaJava(2). The CoqaJava(2) has now around 70% more overhead than Java.

Number of threads	CoqaJava		Java		Ratio
	Mean(second)	std	Mean(second)	std	
1	0.290	0.001	0.189	0.001	1.536
2	0.337	0.006	0.193	0.005	1.744
4	0.354	0.018	0.200	0.004	1.765
8	0.344	0.006	0.199	0.008	1.727
16	0.368	0.011	0.197	0.008	1.867
32	0.395	0.016	0.237	0.014	1.662
64	0.418	0.034	0.241	0.032	1.732

Table 5.5: Java and Optimized CoqaJava(2) on the Puzzle Solver Benchmarks

Figure 5.7 puts the puzzle solver benchmark results on CoqaJava (Table 5.1),

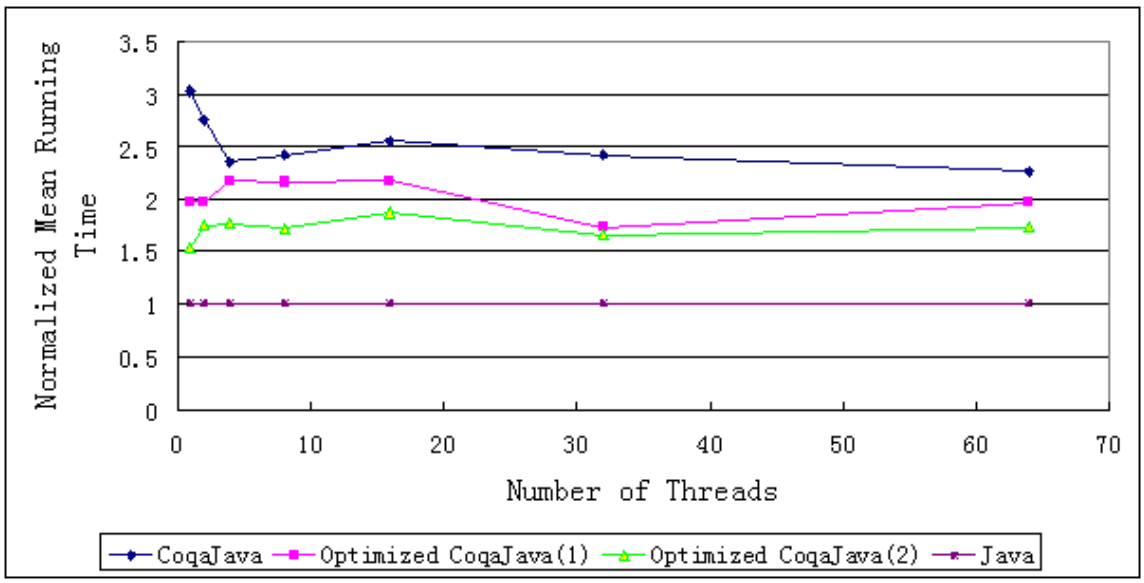

Figure 5.7: Summary of the Puzzle Solver Benchmarks

optimized CoqaJava(1) (Table 5.4) and optimized CoqaJava(2) (Table 5.5) head to head for easy comparison.

Now let see a more interesting case: testing the Java Grande raytracer benchmark in optimized CoqaJava(2). As we had expected, the overhead is greatly reduced after removal of unnecessary locking/unlocking on task-local objects. Table 5.6 and Figure 5.8 show the result of the benchmark on optimized CoqaJava(2), in which overall overhead is reduced to be less than 15% percent, an average of 1.1 times slower than Java. It is a significant improvement compared to that optimized CoqaJava(1) is 17.9 times slower than Java on average.

CoqaJava serves as a proof of concept. It unavoidably suffers extra overhead because it is implemented directly on top of Java. For example, every object capture operation

Number of threads	CoqaJava		Java		Ratio
	Mean(seconds)	std	Mean(seconds)	std	
1	0.253	0.013	0.224	0.016	1.133
2	0.251	0.008	0.235	0.002	1.070
4	0.262	0.017	0.237	0.012	1.107
8	0.267	0.010	0.241	0.004	1.111
16	0.286	0.008	0.253	0.005	1.129
32	0.339	0.012	0.295	0.011	1.147
64	0.435	0.010	0.406	0.023	1.071

Table 5.6: Java and Optimized CoqaJava(2) on the Raytracer Benchmarks

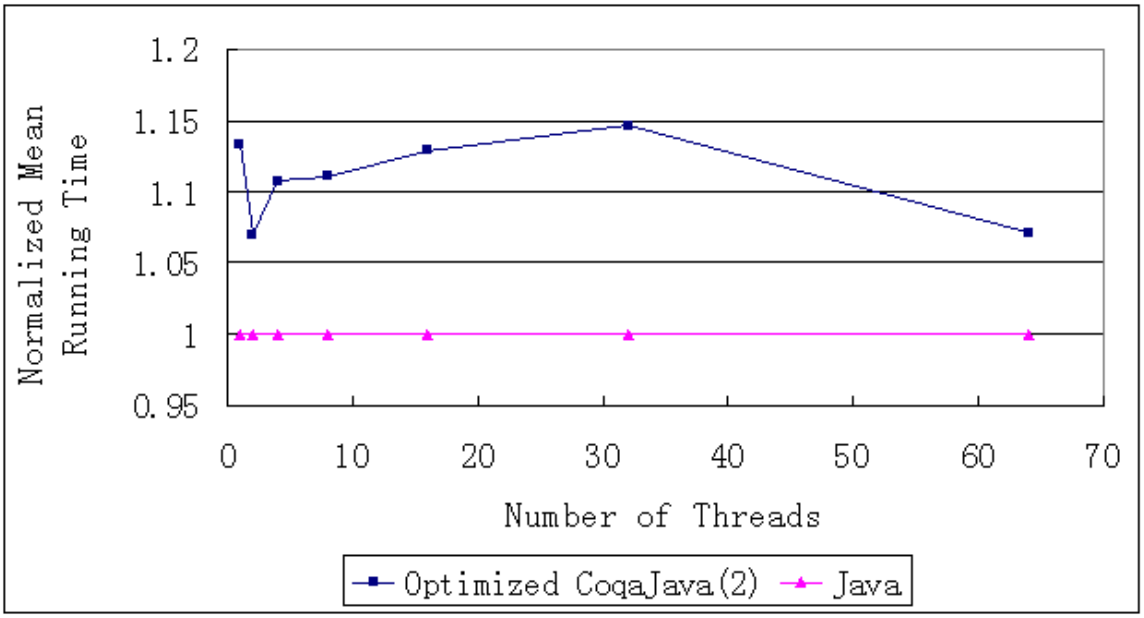

Figure 5.8: Java and Optimized CoqaJava(2) on the Raytracer Benchmarks

in CoqaJava goes through a method invocation. The overhead brought by those method invocations can be huge when a large number of capture operations are involved. However, for such a preliminary implementation, the performance of CoqaJava still comes close to Java, we are optimistic that we can have a more efficient implementation in the future, for instance by building object capture into the lower level Virtual Machine. Section 6.1 gives a more detailed plan on making CoqaJava more efficient.

Chapter 6

Towards a Realistic Language

The focus of the Coqa project is primarily foundational, with more emphasis here on theoretical properties than on implementation features and efficiency. Still, the CoqaJava implementation does show the model is directly implementable and the overhead is not unreasonable. In this chapter, we address some potential extensions which would help to make Coqa a production-quality language. Some of the extensions are specific to the completeness of the language implementation CoqaJava such as library support, while others constitute extensions to the language model itself, such as language constructs for synchronization constraints.

6.1 Tuning Performance

The current implementation CoqaJava is not yet efficient enough since it is still a direct mapping from CoqaJava to Java without serious optimizations. There are many ways that we can further optimize it. For instance, equipping CoqaJava with the ability

to detect immutable objects statically can be very helpful. Immutable objects are objects whose states do not change in their life time once they have been initialized. So immutable objects are always read-only and it is safe to allow concurrent reads on them without locking.

Pushing CoqaJava object capture process into bytecodes is a key long-term plan since that would circumvent the overhead introduced by using Java as an intermediate layer improve the overall performance of CoqaJava.

Currently, Coqa tracks read/write ownership at the object level and every object can be captured by a task individually. In some cases, such an ownership granularity may in fact be *too* fine-grained. For instance, for a clustered data structure consisting of multiple objects, it is more efficient to use a single set of capture flags for the whole structure. Clustering locks into regions so that few actual locks are needed could improve system performance because less locks means less ownership checking and updating.

Many program analysis techniques can also be used in reducing number of locks used in Coqa. [35] develops a mutex inference algorithm efficiently infers a set of locks for shared memory locations. Extending such an algorithm to Coqa will enable the system to be able to distinguish objects that are not accessed by multiple tasks; therefore do not need to be protected by putting them into a task's capture sets. [20] introduces an automatic technique that infers a lock assignment for shared variables and it has an optimization scheme that tries to minimize the conflict cost between atomic sections and the number of locks.

Ownership type systems [54, 17, 46] also can be used to cluster objects and hence reduce the number of locks. The whole-part tree relationship enforced by ownership types

enable a node on the tree to be representative of the subtree rooted from the node. Locking a root is sufficient to lock the whole subtree.

Recent research [7, 56] has developed techniques to avoid using "heavyweight" operating system mutex and condition variables to implement Java monitors, instead they apply more efficient "lightweight" locks whenever possible. Their results show that it is possible to make locking/unlocking of an object fast. Some of those techniques have been implemented and benchmarked on the Java Virtual Machine (JVM). CoqaJava therefore can potentially benefit from a JVM with such an optimized locking scheme indirectly by running as an Java extension or directly by implementing a virtual machine for Coqa.

Clik [16] has a novel work-stealing scheduler in which each processor maintains a work deque of ready threads, and it manipulates the bottom of the deque like a stack. When a processor runs out of work, it steals a thread from the top of a random victim's deque. This greedy scheduling of Clik explores thread locality to achieve substantial performance gain. In Cilk, the average cost of a spawning a thread is only 2-6 times the cost of an ordinary C function call. Currently, tasks in CoqaJava are submitted to a pool of Java threads and CoqaJava does not have control on how a task is scheduled. We believe that CoqaJava can adopt Clik's scheduling algorithm to greatly improve the system performance of CoqaJava.

Task Parallel Library (TPL) by Microsoft [43] provides more expressive parallelism syntax than Cilk via library support. It enables programmers to conveniently express potential parallelism in existing sequential code, where the exposed parallel tasks will be run concurrently on all available processors. Usually this results in significant speedups.

Like Cilk, TPL does not help to correctly synchronize parallel code that uses shared memory. It is still the programmer's responsibility to ensure that certain code can be safely executed in parallel or to use mechanisms such as locks to correctly coordinate parallel computations.

6.2 Exploring a Hybrid System

Coqa uses the pessimistic locking approach to enforce atomicity because it is simply impossible for the optimistic transactional approach to model I/O using a rollback/commit mechanism. However, the choice of applying the locking mechanism on the language semantics level does not mean that the Coqa design rules out transactional approach of STM systems. In fact, we believe it is possible for Coqa to have an underlying implementation which incorporates both locking and rollback techniques.

The reason for exploring a hybrid system is a potential performance gain. The optimistic approach has an advantage in the case where locks are overly conservative, e.g., rollbacks in the optimistic system only happen when contentions actually take place, while the pessimistic system has to block on possible contentions that may not in fact occur at runtime. However, there have been no clear studies that we know of comparing the two different approaches in terms of performance. Generally there 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. Different programming patterns may work better in one system or the other. This is analogous to the relationship between busy-waiting and blocking methods on implementing monitors; although the blocking approach is generally considered to be more efficient, busy-waiting

can be a preferable choice in some cases, for instance, when scheduling overhead is larger than expected wait time or in OS kernels where schedule-based blocking is inappropriate. A good overview of the pros and cons of blocking and rollback appears in [63].

There are many criteria such as the frequency/likelihood of contention that can be used in determining which one is the better choice for implementing a language runtime. For instance, the rollback approach has the advantage of resovling a livelock with high probability after multiple retries. So it can potentially be applied to code blocks that cannot be statically verified to be deadlock free. It would be interesting to consider using static analysis or dynamic profiling techniques to explore structure of a program and then having the compiler or language runtime to choose if an optimistic or a pessimistic approach should be applied.

6.3 Completing CoqaJava

The current CoqaJava implementation is not yet working on all Java features. Moreover, it needs rich libraries to support easy programming. Various engineering efforts that can be applied to CoqaJava to achieve production-level performance were discussed in Section 6.1.

The features left out of CoqaJava for simplicity include static fields, implicit getters/setters, inner classes, and exceptions. None of these are horribly difficult to implement. Here is an outline how some of the more challenging features can be implemented.

Static fields can be added to CoqaJava by treating their accessor methods differently from ones of an instance field: a task needs to capture the `Class` object of the class

where the static field is declared (In Java, every class is associated with a `Class` object at run time). The restriction of having explicitly coded getters/setters can be eliminated by letting the CoqaJava translator automatically generate them for public fields. The CoqaJava translator then can convert all field accesses to method invocation of these generated getter/setters. This way, normal Java field access expressions can be used in CoqaJava. Exceptions are not supported in the prototype, but they are not a particular challenge—unlike optimistic transaction systems where exception handling within a transaction that must rollback is extremely tricky [30], our pessimistic blocking approach can always preserve the exception semantics of an execution. To extend CoqaJava with exception handling, the most crucial thing is to ensure that tasks free their captured objects when they are terminating due to an exception.

Library support is an important aspect of a programming language, and CoqaJava currently lacks such support. Concurrency control classes such as semaphores would enable programmers to do more fine-grained explicit task synchronization.

6.4 Improving Expressiveness

Immutable Objects Section 6.1 discusses using static analysis techniques to detect immutable objects in a program for performance purposes. However, static tools are unavoidably conservative. It is certainly beneficial to enable programmers to declare immutability of an object, especially in Coqa where more lock/unlock operations are performed in order to preserve quantized atomicity. Moreover, allowing programmers to declare immutability of objects can prevent these objects from being accidentally modified at run time. In gen-

eral, it should always be encouraged to make object immutable for safety and performance benefits, especially in a concurrent context. Figure 6.1 gives an extended syntax of Coqa in which a read-only label is used to mark immutable objects. Accordingly, with such an extension, the operational semantics handles immutable object differently so that they are always free and ready to be used by any task.

μ	::=	`r` | `w`	*mutability label*
e	::=	**new** μ $cn(e) \mid \ldots$	*instantiation*
H	::=	$\overrightarrow{o \mapsto \langle cn; \mu; R; W; F \rangle}$	*heap*

Figure 6.1: Dynamic Data Structures for Mutability

The object immutability in Coqa is shallow: fields of an immutable object are immutable, while other objects referenced by the immutable object do not need to be immutable. The semantics of shallow object immutability encourages programmers to declare immutable objects whenever possible. Because to declare an object o immutable, they do not have to ensure immutability of a whole object graph rooted from o. Such an shallow immutability is adequate for Coqa because tasks capture objects one by one, a mutable object referenced by an immutable one cannot escape from being captured even if a task accesses it via the immutable object.

Notification Language-level support for inter-task notification has proved to be useful in practice. Examples along this line include Conditional Critical Regions (CCR) [36], **wait** and **notify**/`notifyAll` primitives in Java, CCR support in STM [31], and `yield_r` in AtomCaml [57].

Coqa does not have such an explicit signal mechanism so concurrent constructs

like semaphores have to use a busy-waiting scheme. Fortunately, it is not very difficult to incorporate a notification protocol into Coqa. The most direct way is to port Java's implementation for these language constructs: the built-in `wait` and `notify/notifyAll` methods for all Java objects. In Chapter 7, we will use the Java's wait/notify protocol directly in CoqaJava as its extension.

Advanced synchronization constructs Coqa can be extended to support *synchronization constraints* declared as part of a method signature. For instance in the classic buffer read/write example [9], we can constrain the `get` method of the `Buffer` to be enabled only when the internal buffer is not empty. Invocations of `get` are blocked (and the `Buffer` object is be captured) until the constraint is satisfied. This extension is not conceptually difficult, and synchronization constraints on objects are also a well studied area, originally in Actors [1, 2] and later in Polyphonic C# [9]. Adding synchronization constraints does complicate matters on systems with class inheritance, but standard solutions do exist. [50] gives an in-depth overview of this topic.

6.5 Deadlock Discovery

Deadlocks are undeniably the biggest disadvantage of the locking approach, and Coqa unavoidably runs into this problem because Coqa uses the locking mechanism to ensure atomicity. First of all, Coqa still largely reduces programmers' burden of developing concurrency applications in spite of the deadlock issue, because it has shifted tasks of dealing with all different types of problems in a concurrent context to just focusing on deadlock detection and the code refactoring needed to fix it. For instance, using Java to

program a multi-threaded application means that programmers need to take care of data races, mutual exclusion, atomicity and deadlocks all by themselves. In Coqa all other issues besides deadlocks are addressed, and so programmers can tightly focus on deadlocks.

There are two forms of deadlock arising in Coqa. The first form was described in Section 2.1: the deadlock inherent in two-phase locking. The second form is the standard cyclically dependent deadlock. The first form of deadlock can be avoided by declaring a class `exclusive`, while the second form of deadlock can be removed by using subtasking to break a deadlock cycle.

Developing quality deadlock discovery tools for Coqa for both compile time and run time analysis is important to alleviate a programmer's debugging burden. There has been a great deal of research focusing on solving the deadlock problem using various approaches. Deadlock avoidance avoids actions that may lead to a deadlock state. To make correct decisions, a system must know in advance at any time the number and type of available/allocated resources. However, most practical systems do not have such preconditions. For instance, it is generally impossible for Coqa, an object-oriented language model, to know in advance how many objects will be created at run time. Runtime deadlock detection usually involves a system tracking resource allocation and processing state and determining if there is a deadlock in the system. But it is impossible to design and implement a general recovery strategy when a deadlock is detected at run time.

It is our belief that deadlocks can be addressed by deadlock prevention, a static approach preventing deadlock conditions from holding. Hence, deadlocks in Coqa can be thought as type errors and static deadlock prevention algorithms work like a type system

to catch these errors before run time. So static analysis techniques and tools that ensure deadlock freedom in Coqa are an important component to the success of Coqa.

There has been extensive research on detecting deadlocks statically and we plan to explore the possibility of porting existing techniques to Coqa. [11] proposed an ownership type system to ensure locks can only be acquired in a non-cyclic way. Equipping Coqa with such a type system can help programmers to type-check deadlocks in their applications. Model checking systems such as Bandera [18] and SPIN [37] have been used to verify deadlock freedom and they could be used by Coqa as well. Coqa can also utilize the lock-order graph commonly used for static deadlock detection systems. Jlint [39] is one of the pioneering works in this field. [62, 65] extend the method to use a lock-order graph and context-sensitive lock sets, which makes the analysis more sophisticated and complete. RacerX [21] is a flow-sensitive, context-sensitive interprocedural analysis to detect both race conditions and deadlocks for C and requires explicit programmer annotations. These techniques should be applicable to Coqa.

Chapter 7

Concurrent Programming in CoqaJava

In this chapter, we demonstrate the ease of programming of CoqaJava via common concurrent programming patterns. Java programmers should be able to adapt to the CoqaJava programming environment easily because CoqaJava has nearly identical syntax as Java except the different semantics of the three types of message send. Recall that methods in CoqaJava can be either **public** or **private**. The two method modifiers have the same meaning as in Java. Instance fields in CoqaJava are always accessible only via their getters/setters. In order to give clear and concise code examples, the programming patterns discussed in this chapter are coded in CoqaJava with a few minor syntax abbreviations. Field access can be performed as in Java, without using getter/setter methods. A field with no access modifier is by default a **private** field with **private** getter/setter, while a field declared as **public** is a **private** field with **public** getter/setter methods. We also borrow

some data structure classes such as `ArrayList` from Java for easy programming.

7.1 Powerful Built-in Atomicity in CoqaJava

In Section 1.1.3, we used an example of race-free but incorrect program (Figure 1.3) to explain the importance of atomicity. We use the same example in this section to show how program correctness can be easily guaranteed in CoqaJava because of its built-in atomicity.

Consider the Java example below; there is no data race here. However, erroneous output can be generated if one thread, td_1, executes line `L2` and `L3`, then another thread, td_2, executes `L1`, then when td_1 continues with line `L4`, the value of `balance` has already been changed by td_2 and therefore it would write an inconsistent value to the `balance` field. It is not to hard to see that this can be a hard-to-detect bug in a real application especially when the size of the application is big.

```
        class Bank {                              //Java code
            int balance;
            void synchronized deposit(int n) {
(L1)            balance = balance + n;
            }
           void withdraw(int n) {
(L2)           int r = read();
               synchronized(this) {
(L4)               balance = r - n;
               }
           }
           int synchronized read(){
                int r;
(L3)            r = balance;
                return r;
           }
        }
```

Porting the above Java example into CoqaJava yields the following nearly identical

code with only Java's **synchronized** keywords removed. The buggy execution path cannot happen in this CoqaJava program. Assume that we have two concurrent tasks t_1 and t_2 corresponding with the td_1 and td_2 in the Java example. When t_1 executes line `L2` and `L3`, the task will capture the current `Bank` object. At this point, if t_2 reaches line `L1`, it cannot proceed and has to block on the same `Bank` object until t_1 finishes executing line `L4` and frees the `Bank` object. As a result, the field `balance` will never be left in an inconsistent state.

```
        template Bank {                          //CoqaJava code
            int balance;
           void deposit(int n) {
(L1)            balance = balance + n;
            }
           void withdraw(int n) {
(L2)           int r = read();
(L4)            balance = r - n;
           }
           int read(){
                int r;
(L3)            r = balance;
                return r;
           }
        }
```

CoqaJava eliminates these types of bugs in a very elegant way. In Section 7.3.2 we will discuss how a data race bug is nicely handled in CoqaJava.

7.2 Built-in Concurrency in CoqaJava

`ConcurrentHashMap` in the `java.util.concurrent` package uses fine-grained locking to support full concurrency of retrievals to minimize unnecessary lockings. Since CoqaJava has the read/write locking scheme built-in, such maximal concurrency can be

easily coded in CoqaJava without explicit use of locking or synchronization which Java library programmers have to deal with. The following code gives an illustration how `ConcurrentHashMap` could be implemented in CoqaJava. `HashTable` used here is given in Figure 2.2. The basic strategy is to keep a set internal hash tables, each of the internal hash table is itself a concurrently readable table. No concurrent tasks would block in line `L1` since they all read the `ConcurrentHashMap`. There is no explicit lock acquire/release in method `put` as required by Java, but tasks will be blocked in `L2` when they try to write an internal hash table.

```
          template ConcurrentHashMap {
              HashTable[] hashtables ;
              ConcurrentHashtable(int n) {
                 hashtables = new HashTable[];
                 for(int i =0 ; i < n ; i++) {
                    hashtables[i] = new HashTable();
                 }
               }

              public Object get(Object key) {
                  int hash = this⇒hash(key);
(L1)              return hashtableFor(hash)⇒get(key);
              }

              public void put(Object key, Object value) {
                  int hash = this⇒hash(key);
(L2)              return hashtableFor(hash)⇒put(key,value);
              }
              int hash(int x) {
                  int h = x.hashCode();
                  //compute h;
                  return h;
              }
              Hashtable hashtableFor(int hash) {
                  int index = ... //compute index
                  return (HashTable) hastables[index];
              }
          }
```

The `ConcurrentHashMap` example shows CoqaJava has natural expressiveness for concurrency that is needed for implementing many high throughput concurrent data structures.

7.3 Concurrent Programming Patterns

While coding common concurrent patterns in CoqaJava produces code very similar to Java code, the newly introduced operators -> and => have nontrivial implications in program design and performance. We first use a CoqaJava implementation of the classic producer/consumer pattern to illustrate the usage of the two new operators, and then give actual coding examples of various common concurrent programming patterns including oneway messages, completion callbacks, futures, fork/join, concurrent control utilities such as semaphore, and barriers [42]. All the CoqaJava implementations of the patterns are based on the Java implementation described in [42]. The examples in this chapter are intended to serve as illustration only, and are not production quality programs. For example, we omit exception handling code, and may use infinite loops where necessary for simplicity. Some patterns discussed in [42] are trivial in CoqaJava. For instance, fully synchronized objects and synchronized aggregate operations are patterns built into CoqaJava. From these examples, we can see that adapting from Java programming to CoqaJava programming is relatively straightforward transition.

7.3.1 Producer/Consumer

Below we give a CoqaJava implementation of the common producer/consumer pattern: producers place dummy objects into a shared queue, consumers remove objects from the same shared queue, and they run concurrently in separate tasks. A `Semaphore` object is used to keep consumers wait if the queue is empty. Let us assume for now that the `Semaphore` object used in this example is a native object that no task can capture and which provides the functionality of a counting semaphore. We will discuss how a `Semaphore` can be implemented in CoqaJava in Section 7.3.6.

Asynchronous messages sent at `L1` and `L2` via `->` create tasks that run producer and consumer code respectively. The `->` invocation is non-blocking and encapsulates a series of operations such as task creation and task initialization internally. Note that no **synchronized** keyword is used anywhere in the code, even though producers and consumers access the shared `Queue` objects. This is because synchronization is provided implicitly by CoqaJava: internal locking is performed when the `Queue` object is actually accessed in method `get` (line `L5`) and `put` (line `L6`). Lines `L3` and `L4` are the programming points where `=>` is used to create subtasks that access the shared `Queue` object.

As we can see from this example, coding in CoqaJava and in Java are not so different in syntax, but we have to make conscious choice about using regular (.) invocation and `=>` invocation on shared objects. For example, a consumer task uses a subtask created in line `L3` to access the shared `queue` object so that the object would be freed when the subtask returns. If a (.) invocation were used here, the `queue` object would be captured by the current consumer task until the whole `work` method finishes.

```
template ProducerConsumerBench {
    ...
    public void main(String[] args) {
        int numOfProducer = ...;
        int numOfConsumer = ...;
        Queue queue = new Queue(10);
        Producer p = new Producer(queue);
        for(int i=0; i < numOfProducer; ++i) {
(L1)          p->work();
        }
        Consumer c = new Consumer(queue);
        for(int i=0; i < numOfConsumer; ++i) {
(L2)        c->work();
        }
    }

}

template Consumer {
    Queue queue;
    ...
    public Consumer(Queue queue) {
        this.queue = queue;
        ...
    }
    public void work() {
        while(1) {
(L3)        Item item = queue=>get();
            // ... do someting with item
        }
    }
}

template Producer {
    Queue queue;
    ...
    public Producer(Queue queue) {
        this.queue = queue;
        ...
    }
    public void work() {
        while(1) {
            int value = compute();
```

```
(L4)                queue ⇒ put(value);
                }
            }
            public int compute() {
                int value;
                ...
                return value;
            }
        }

        template Item {
            public int value;
            public Item(int v) {
                value = v;
            }
        }

        template Queue {
            ArrayList<Item> contents;
            Semaphore guard;
            Queue(int init) {
                contents = new ArrayList<Item>(init);
                guard   = new Semaphore(0);
            }
(L5)        Item get() {
                guard.acquire();
                Item item = contents.remove(0);
                return item;
            }
(L6)        void put(int value) {
                Item item = new Item(value);
                contents.add(item);
                guard.release();
           }
           ...
        }
```

7.3.2 Oneway Messages

An oneway message is a pattern in which a host object issues a message to one or more recipients without depending on the consequences of that message [42]. Open calls

and thread-per-message are examples of this pattern.

As stated in [42], the use of open calls does not address how to introduce concurrency into a system but rather eliminates bottleneck points surrounding a giving host. As in the following Java code, the request acceptance rate of the `OpenCallsHost` is bounded only by the time it takes to update local state so that the host can accept requests from clients running in different threads even if the `helper.handle` call is time-consuming.

```
class State {                          //Generic Java code sketch
    private long x;
    private long y;
    public void update(...) {
        x = ...;
        y = ...;
    }
}

class OpenCallsHost {
    protected State localState;
    protected Helper helper = new Helper();
    ...
    public synchronized void updateState(...) {
        ...
        localstate.update(...) ;
        ...
    }
    public void  req(...) {
        this.updateState(...);
        helper.handle(...);
    }
}
```

Notice that although the Java code above looks correct even to an experienced Java programmer, there is a serious trap hiding in the snippet, which can manifest itself as a hard-to-detect bug later in a concurrent context. Specifically, the `updateState` method is declared as **synchronized** so only one thread can call the `update` method on the

`localState` object. However, the fact that the `localState` field is declared as **protected** opens a door to threads that have access to the object but not necessarily access it via the provided synchronized `updateState` method. Instead, these threads can directly call the `update` method on the `localState` object. Because the `update` method is not **synchronized**, it is possible for two threads to enter this method, leading to inconsistent values of `x` and `y`, a typical data race bug.

Coding the same open calls example in CoqaJava looks like the following. It achieves the same goal of the Java code snippet by using ⇒ to update the local state, but it avoids the potential race condition. According to the semantics of CoqaJava, tasks arriving at the `update` method of the `localState` object, no matter whether via calling the `updateState` method of the `OpenCallsHost` or by directly calling the `update` method on the `localState` object, execute the method atomically, which always leaves `x` and `y` with consistent values. This shows the strength of Coqa: the atomicity property is always guaranteed for *all* objects directly or indirectly accessed at run time by a method.

```
template OpenCallsHost {          //Generic CoqaJava code sketch
    State localState;
    Helper helper = new Helper();

    public void  updateState(...) {
        ...
        localState.update();
        ...
    }
    public void  req(...) {
        this ⇒ updateState(...);
        helper.handle(...);
    }
}
```

In the Java thread-per-message example below, concurrency is introduced by issuing a message in its own Java thread. In the CoqaJava code, -> is instead used for the equivalent task-per-message. This example shows how CoqaJava has simpler syntax than Java.

```
template ThreadPerMessageHost {       //Generic Java code sketch
    protected long localState;
    protected Helper helper = new Helper();

    public synchronized void upateState(...) {
        localState = ...
    }
    public void req(...) {
        updateState(...);
        new Thread(new Runnable() {
          public void run() {
             helper.handle(...);
          }
        }).start();
    }
}
```

```
template ThreadPerMessageHost {   //Generic CoqaJava code sketch
   long localState;
   Helper helper = new Helper();

   public void upateState(...) {
       localState = ...
   }
   public void req(...) {
       this ⇒ updateState(...);
       helper -> handle(...);
   }
}
```

7.3.3 Completion Callbacks

Completion callbacks are a pattern where a host requests a worker to do something concurrently and requires the notification of the worker's completion. An example of how this pattern works in CoqaJava is given below. The host task creates a worker task at `L1` using asynchronous operator `->`. The `Worker` calls back `Host` at `L2` using synchronous operator `=>`. The `=>` is recommended here if the `callback` method in `Host` accesses (reads or writes) its instance fields so the `Worker` task can release the capture of the `Host` object right after the completion of the callback method instead of the completion of the worker task.

```
            interface  CallCompletionInterface {
                void callback(...);
            }
            template Host implements CallCompletionInterface {
                Worker worker;
                public Host() {
                    worker = new Worker();
                }
                public void start(...) {
                    ...
(L1)                worker -> doSomething(this);
                    ...
                }
                public void callback(...) {
                    ...
                }
             }
             template Worker {
                 CallCompletionInterface observer;
                 public void doSomething(CallCompletionInterface c) {
                     observer = c;
(L2)                 observer => callback(...);
                     ...
                 }
             }
```

7.3.4 Futures

Essentially, futures are place holder objects for some unknown data. The data is unknown because the process of computing the data has not yet completed. With the use of futures, a task can start some other tasks for doing concurrent computations and then fetch results of those tasks later when the results are actually needed for its own computation. If at the fetching point, the data is still not ready, the calling task must block until the data become available.

Futures are a useful pattern in CoqaJava. Recall that there is no return value for sending an asynchronous message using `->` syntax. This means in CoqaJava a child task cannot return a value to its parent task directly. With the help of futures, a return value from a child task can be encoded.

Below shows an example implementation of futures in CoqaJava. A `Result` object encapsulates the data of a `Future` object. The `Worker` class is responsible for computing the data of a `Result` object. The task starting from the `start` method of a `Host` object first creates a worker task executing on a `Worker` object with a reference of a `Future` object (line `L1`). Then, the worker task captures the future object after calling the `set` method on the object at `L3` using a (.) invocation. The worker task keeps its ownership of the `Future` object until its completion. The (.) invocation at `L3` is crucial here to force the host task to block at `L2.1` or `L2.2`. The difference between `L2.1` and `L2.2` is that the `Future` object will be released by the host task after the statement of `L2.2` so that it can be passed to other worker tasks if necessary, but it will remain captured by the host task with the execution of `L2.1`.

```
            template Result {
                public int value1;
                public float value2;
                ...
            }

            template Future {
                Result result;
                public Result get() {
                    //the host task blocks here until
                    //the worker task finishes with this object
                    return result;
                }
                public void set(Result v) {
                    // the worker task calls this to
                    // get a write lock on this object
                    // until its completion
                    result = v;
                }
            }

            template Host {
                Future future;
                public Host() {
                    future = new Future();
                }
                public void start(...) {
                     ...
                     Worker worker = new Worker(future);
(L1)                 worker->work(...);
                     //...do some other work
(L2.1)               Result r = future.get();
                     // or
(L2.2)               Result r = future⇒get();
                     //...continue with the result
                }
            }

            template Worker {
                Result result;
                ...
                public Worker(Future future) {
                    result = new Result();
```

```
(L3)                future.set(result);
                }
                public void work(...) {
                    int v1;
                    float v2;
                    // ... compute v1 and v2
                    result.value1 = v1;
                    result.value2 = v2;
                }
            }
            template Main {
                public void main(String[] arg) {
                    (new Host()).start();
                }
            }
```

Using Java to realize futures requires programmers to use the explicit wait/notify scheme with extra coding as shown below.

```
            class Future {                                  //Java code
                Result result;
                boolean ready;

                public synchronized Result get() {
                    while (!ready)
                        wait();
                    return result;
                }
                public synchronized void set(Result v) {
                    ready = true;
                    result = v;
                    notifyAll();
                }
            }
```

7.3.5 Fork/Join

The fork/join pattern is often used to exploit the parallelization of task solving algorithms – a task is decomposed into a set of child tasks and the results are later combined

to reach the solution to the original task.

The following example uses the futures pattern and refers to the classes such as `Future`, `Result` and `Worker` defined in the previous section. The host task creates a worker task for each problem in line `L1` and then collects results computed by the worker task in line `L2` where the host task blocks on each `Future` object until the associated worker task finishes its computation.

```
        template Host {
            ...
            public void compute(Set<Param> problemSet) {
                ListSet<Future> futures = new ListSet<Future>();
                Iterator it = problemSet.iterator();
                while(it.hasNext()) {
                    // create a task for each problem
                    Param problem = it.next();
                    Future future = new Future();
                    Worker worker = new Worker(future);
(L1)                worker->work(problem);
                    futures.add(future);
                }
                combineResults(futures);
            }
            private void combineResults(Set<Future> futures) {
                ListSet<Result> results = new ListSet<Result>();
                Iterator it = futures.iterator();
                while(it.hasMore()) {
                    Future future = it.next();
(L2)                Result r = future.get();
                    results.add(r);
                }
                // we have all the results now
                // do something with the results ...
            }
        }
        template Main {
            public void main(String[] arg) {
                (new Host()).compute();
            }
        }
```

7.3.6 Concurrent Control Utilities

In this section, we take a in-depth look at the patterns of some low-level synchronization utilities such as `Semaphore`, `CountDownLatch` and `CyclicBarrier`. They encapsulate certain intra-tasking synchronization protocols with various convenient features.

Semaphore Below, we give an illustration how such a `Semaphore` class would be implemented using a busy-waiting scheme.

```
            template  Semaphore {
                int  permits;

                public Semaphore(int permits) {
                    this.permits = permits;
                }

                public void acquire() {
                    boolean success = false;
                    while(success == false) {
(L1)                    success = this ⇒ dec();
                    }
                }
                private boolean dec() {
                    if(permits > 0) {
                        permits = permits - 1;
                        return true;
                    }else {
                        return false;
                    }
                }

                public void release() {
(L2)                this ⇒ inc();
                }
                private void inc() {
                    permits = permits + 1;
                }
            }
```

The important property of a `Semaphore` object is that it is essentially a shared object whose internal state is the channel for intra-task communication. Therefore, the implementation should prevent a `Semaphore` object from being captured by one task except for a short duration of updating the state of the object. As shown in the above implementation, the `Semaphore` class makes sure that a `Semaphore` object's `permits` field is updated in a mutually exclusive manner by subtasks in line `L1` and `L2`. With such an implementation, a `Semaphore` object can be properly invoked via (.) notation, as shown in the producer/consumer example at the beginning of this chapter.

Busy-waiting semaphore wastes system resources. A more efficient `Semaphore` class can be implemented if there is an intra-task notification mechanism in CoqaJava. For now, we directly adopt `wait` and `notify` (or `notifyAll`), the two signal primitives provided by Java into CoqaJava: like objects in Java, every object in CoqaJava has built-in methods `wait` and `notify`/`notifyAll`. When a task calls `wait`, it goes into a blocking state waiting for being notified by other tasks via `notify`, at the same time, it relinquishes its ownership of the object it calls `wait` on.

The new implementation of the `Semaphore` class below does not differ much from a Java one. The `Semaphore` class is actually a wrapper class enclosing a `SemaphoreImpl` object named `impl`. The actual semaphore protocol is realized in the `SemaphoreImpl` class. The `Semaphore` class delegates calls to the inner `impl` object via `=>` which ensures the `impl` object is not captured by a single task so multiple tasks can interact via mutating the state of the `impl`.

```
template  Semaphore {
    SemaphoreImpl impl;
    public Semaphore(int permits) {
```

```
        impl = new SemaphoreImpl(permits);
    }

    public void acquire() {
        impl ⇒ acquire();
    }
    public void release() {
        impl ⇒ release();
    }
}
template exclusive SemaphoreImpl {
    int permits;
    public SemaphoreImpl(int permits) {
        this.permits = permits;
    }

    void acquire() {
        while(permits <= 0) wait();
         --permits;
    }
    public void release() {
        ++permits;
        notify();
    }
}
```

CountDownLatch and CyclicBarrier A latch variable is one that eventually reaches a value from which it never changes again. It enables a task to wait for signals from a set of other tasks. By calling the `await` method of a `CountDownLatch` object, a task enters a wait state until a pre-defined number of tasks each call the `countdown` method of the same `CountDownLatch` object. The `CountDownLatch` observes the same implementation technique used in `Semaphore` class.

```
template CountDownLatch {
    LatchCounter lc;
```

```
    public CountDownLatch(int c) {
        lc = new LatchCounter(c);
    }

    public int countdown() {
         lc ⇒ countdown();
    }
    public int await() {
         ls ⇒ await();
    }
}

template exclusive LatchCounter {
    int parties;
    public LatchCounter(int parties) {
        this.parties = parties;
    }

    public void countdown() {
        --parties;
        if(parties == 0) notify();
    }
    public void await() {
        if(parties != 0) wait();
    }
}
```

A cyclic barrier is a more advanced concurrent device that can be used to synchronized tasks to proceed together from a pre-set starting point in a cyclic way. It is initialized with a fixed number of parties that will be repeatedly synchronized. The following `CyclicBarrier` shows a simple implementation of this type of control scheme. By calling the `await` method of a `CyclicBarrier` object, each task will be forced to wait until a pre-defined number of tasks have invoked the method, then, the `CyclicBarrier` object resets itself for the next iteration.

```
template CyclicBarrier {
    CBInternal cbi;
```

```
    public CyclicBarrier(int c) {
        cbi = new CBInternal();
    }


    private int await() {
        return cbi.await();
    }
}
template exclusive CBInternal {
    int parties;
    int count;   //parties currently being waited for
    int resets;  //times barrier has been tripped

    public CBInternal(int c) {
        count = parties = c;
        resets = 0;
    }

    public int await() {
        int index = --count;
        if(index > 0) {    //not yet tripped
           int r = resets; //wait until next reset
           do { wait();} while (resets == r);
        }else {             //trip
           count = parties;    //reset count for next time
           ++resets;
           notify();       //cause all other parties to resume
        }
        return index;
    }
}
```

7.3.7 Barriers

In this section, we demonstrate a more complex pattern that takes advantage of the `CountDownLatch` and `CyclicBarrier` we just discussed.

In this pattern, a set of tasks work together on an iterative algorithm. Tasks enter each iteration together and wait for all tasks to complete at the end of the iteration. The

algorithm usually finishes at some predefined number of iterations or when some convergence condition is reached.

In the following example, the host task starting from the `start` method of the `Host` object divides a given problem into a given number of segments and uses `->` in line `L1` to create child tasks for each of the segments. Each segment task runs concurrently and waits for others to finish at `L3` before entering the next iteration. A segment task may access some shared objects during each iteration, and such accesses are recommended to use `=>` operator as what the example does at `L2`.

```
        template Segment {
            CountDownLatch done;
            public Segment(CountDownLatch done) {
                this.done = done;
            }
            public void update(CyclicBarrier bar, Problem p, int itr) {
                bar.await(); // start together
                for(int i =0; i < itr; i++) {
                    ...
(L2)                p => dosomething();
                    ...
(L3)               bar.await();// iterate together
                }
                done.countDown();// exit together
            }
        }

        template Host {
            public void start(Problem p, int numOfSegt, int numOfItr) {
                CountDownLatch done = new CountDownLatch(numOfSegt);
                Segment s = new Segment(done);
                CyclicBarrier bar = new CyclicBarrier(numOfSegt);
                for(int i = 0; i < numOfSegt; i++) {
(L1)                s -> update(bar, p, numOfItr);
                }
                done.await();
            }
        }
```

```
template Main {
    public void main(String[] arg) {
        (new Host()).start();
    }
}
```

Chapter 8

Related Work

We now review related work, with a focus on how atomicity is supported, and how concurrent object models are constructed.

8.1 Actors-Like Languages

Actors [1, 2] provide a simple concurrent model where each actor is a concurrent unit with its own local state. Over the decades, Actors have influenced the design of many concurrent object-oriented programming models, such as ABCL [66], POOL [4], Active Objects [14], Erlang [6, 22], the E language [51], and Scala [29].

As discussed in Section 1.2.2, Actors are a model more suited to loosely-coupled distributed programming, not tightly coupled concurrent programming. So, when Actor languages are implemented, additional language constructs (such as futures, and explicit continuation capture) typically 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. Moreover, in contrast to Coqa's ubiquitous and deep atomicity, Actors have ubiquitous but shallow atomicity as discussed in Section 1.2.4 in the sense that atomicity is a per-actor-per-message property in the Actor model.

In Scala [29], 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 [29] 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.

Erlang [22] is a concurrent programming language following the Actor model. Its main strength is to support distributed applications in which asynchronous messaging is a natural fit. Similar to [29], Erlang programmers have to encode the Pid (process identifier) in messages in order to properly simulate synchronous message passing.

E language [51] has an actor-based messaging. It also shares the local synchronous messaging (immediate calls in E) and asynchronous nonlocal messaging (eventual send in E) aspect of Coqa, with the nonlocal messaging being distributed. The when-catch construct of E provides an elegant continuation capture syntax which allows continuation of futures (promises in E) to nest within a method instead of being written as a distinct new method. Each object in E belongs to exactly one vat (analogous to a process). Each vat has a single thread of execution, a stack frame, and an event queue. Arriving messages are placed into the vat's event queue and then processed one by one atomically. However, this indeed shows that E language is essentially a single-threaded language model in which atomicity is achieved at expense of a loss of expressivity of multithreaded programming.

8.2 Software Transactional Memory

The property of atomicity is commonly discussed in various STM systems: systems with optimistic and concurrent execution of code that should be atomic, and the ability to rollback when contention happens. Early software-only systems along this line include [59, 33]. Programming language support for STM started with Harris and Fraser [31], and then appeared in many languages and language extensions, such as Transactional Monitors [64] for Java, Concurrent Haskell [32], AtomCaml [57], Atomos [13], X10 [15], Fortress [3] and Chapel [19].

STM systems vary on the atomicity semantics they support: weak atomicity and strong atomicity [13]. As we have discussed in Section 1.1.3, in proposals with weak atomicity, transactional isolation is only guaranteed between code running in transactions, but not between transactions and non-transactional code. Such a strategy can lead to surprising results if non-transactional code reads or writes data that is part of a transaction's read or write set, and also dwindles the appeal of the atomicity property in the first place: reasoning about atomic blocks of programs is sound only if transactional code and non-transactional code are perfect disjoint on shared resources. A large number of transactional memory systems [34, 5, 59, 33, 31, 64] and languages [15, 3, 19] only conform to the semantics of weak atomicity. Strong atomicity is supported by [32, 57, 13]. In Coqa, since all method invocations are either part of a task (o .$m(v)$), starting up a new task (o ->$m(v)$), or starting up a new subtask (o =>$m(v)$), no code is *ever* running in a non-atomic mode, and hence strong atomicity is trivially preserved.

From the language abstraction perspective, the common abstraction for atomicity

support in OO languages using the STM approach is to define a program fragment as an `atomic` block. As we discussed earlier, this abstraction does not align well with the fact that the base language is an OO language. In Coqa language, quantized atomicity is built deeply into the object model and guaranteed for every complete execution of every method.

Subtasking in Coqa is similar to the open nesting supported by Atomos. But their semantics are not exactly the same. In Atomos, a child transaction is able to commit all objects it has written. If we ignore the difference between the optimistic vs. pessimistic approach, then Atomos's open nesting strategy could be viewed as a subtask in our model that early releases all objects it has worked on, including those belonging to all its parents. This obviously would lead to surprising results. Indeed, the Atomos authors claimed (in Section 3.3 of the same paper): "... changes from the parent transaction can be committed if they are also written by the open-nested child transaction, seemingly breaking the atomicity guarantees of the parent transactions. However, in the common usage of open-nested transactions, the write sets are typically disjoint. This can be enforced through standard object-oriented encapsulation techniques." In other words, the open nesting mechanism of Atomos in itself is unsound, and it relies on either "typical" programming patterns or extra language enforcement. In fact, we have show in Section 2.2.2, the disjoint assumption is unnecessarily too strong in Coqa.

Open nesting as a language feature is studied in [52, 53]. In [53], 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. As another difference, [53] asks the method writer to decide whether the method should be invoked as an open nesting or not. Coqa however gives the decision making to the method invoking party. Such a design gives flexibility to programming.

STM systems have different philosophy on the default atomicity mode a system should have: atomicity is selective and should only be applied to some specific fragment of a program. Taking such a standpoint is an unavoidable choice as to the STM methodology itself. First, I/O is impossible to be rolled back as an transaction and the overhead of logging the state of transactions makes the system suffer increasing number and size of transactions.

8.3 Other Language Models

Argus [45] 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 dimension. Argus is a transaction-based system in that actions can rollback, but unlike STM systems they use a combination of locking and rollback. 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 has a *subaction* mechanism which is a form of closed nesting. 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 Coqa subtask does. Subactions of Argus can only inherit read/write locks from its ancestors but are not able to acquire extra locks on their own.

The Java language has a language construct **synchronized** to enforce only shallow mutual exclusion, as discussed Section 1.1.3. It expects great efforts from programmer to ensure atomicity, a property better captured by atomicity [23]. Java 1.5 provides a new package `java.util.concurrent.atomic` allows programmers to access low-level data types such as integers in an atomic fashion. This effort does not overlap with our desire of building high-level programming constructs with atomicity properties.

Guava [8] 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 [49] is a similar design in spirit.

In [61], a data-centric approach is described for programmers to express their needs for data race and atomicity. Programmers add annotations to group a set of fields, indicating that the elements of such as set must be updated atomically by a unit of work coinciding with method bodies. This approach can be elegant in expressing single-object properties, but if many objects are involved, programmers need to have a clear picture about field aliasing among all those objects.

Chapter 9

Conclusion

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 object messaging, which account for (normal) local synchronous message send via (.), concurrent computation spawning via asynchronous message send by -> and subtasking via synchronous nonlocal send using =>. 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. Quantum of a task is demarcated by task/subtask creation via ->/=>. As long as programmers use them consciously in their programs, the number of interleavings among concurrent tasks is reduced significantly, which facilitates reasoning concurrent applications tremendously.

We then extend the study of quantized atomicity to I/O in KernelCoqa$_{fio}$ and

KernelCoqa$_{rio}$. KernelCoqa$_{fio}$ is the common way how I/O is modeled in existing language systems and it has the limited form of quantized atomicity for I/O operations: one quantum per I/O operation. While the KernelCoqa$_{rio}$ illustrates a stronger atomicity property: a series of I/O steps of a task can be grouped into one atomic quantum and be guaranteed to happen without interleavings. Furthermore, we demonstrate that in KernelCoqa$_{rio}$ certain form of quanta can be joined together to form a bigger atomic zone, resulting an even stronger atomicity property. Moreover, we give an in-depth analysis of how the inter-relationship between I/O objects affects atomicity, an interesting problem that has not been carefully studied before as to our knowledge.

We justify our approach by implementing CoqaJava, a Java extension incorporating all of the KernelCoqa features. A series of benchmarks were conducted to test the performance of CoqaJava. With these benchmark results, we believe the overhead of Coqa for performing more object locking/unlocking can be minimized and does not seriously affect the practicability of Coqa as a general programming language model.

Bibliography

[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] America, P., de Bakker, J., Kok, J. N., and Rutten, J. J. M. M. Operational semantics of a parallel object-oriented language. In *POPL '86* (New York, NY, USA, 1986), ACM Press, pp. 194–208.

[5] Ananian, C. S., Asanovic, K., Kuszmaul, B. C., Leiserson, C. E., and Lie, S. Unbounded transactional memory. In *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture* (Feb 2005), pp. 316–327.

[6] 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.

[7] Bacon, D. F., Konuru, R., Murthy, C., and Serrano, M. Thin locks: featherweight synchronization for java. In *PLDI '98* (New York, NY, USA, 1998), ACM Press, pp. 258–268.

[8] 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.

[9] Benton, N., Cardelli, L., and Fournet, C. Modern concurrency abstractions for c#. *ACM Trans. Program. Lang. Syst. 26*, 5 (2004), 769–804.

[10] Blundell, C., Lewis, E., and Martin, M. Deconstructing transactional semantics: The subtleties of atomicity, June 2005.

[11] Boyapati, C., Lee, R., and Rinard, M. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA '02* (New York, NY, USA, 2002), ACM Press, pp. 211–230.

[12] Campione, M., Walrath, K., and Huml, A. *The Java Tutorial Third Edition: A Short Course on the Basics.* Prentice Hall PTR, 2000. http://www.csse.uwa.edu.au/programming/javatut/index.html.

[13] 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).

[14] Caromel, D., Henrio, L., and Serpette, B. P. Asynchronous and deterministic objects. In *POPL '04* (New York, NY, USA, 2004), ACM Press, pp. 123–134.

[15] Charles, P., Grothoff, C., Saraswat, V. A., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA 2005)* (October 2005), pp. 519–538.

[16] cilk. Cilk. Available at `http://supertech.csail.mit.edu/cilk/`.

[17] Clarke, D. *Object Ownership and Containment.* PhD thesis, University of New South Wales, July 2001.

[18] Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Păsăreanu, C. S., Robby, and Zheng, H. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering* (2000), pp. 439–448.

[19] Cray Inc. Chapel Specification, 2005.

[20] Emmi, M., Fischer, J., Jhala, R., and Majumdar, R. Lock allocation. In *POPL '07* (2007).

[21] Engler, D., and Ashcraft, K. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2003), ACM Press, pp. 237–252.

[22] Erlang. Erlang. Available at `http://www.erlang.org/`.

[23] 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.

[24] Flanagan, C., and Qadeer, S. Types for atomicity. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation* (New York, NY, USA, 2003), ACM Press, pp. 1–12.

[25] Freund, S. N., and Qadeer, S. Exploiting purity for atomicity. *IEEE Trans. Softw. Eng. 31*, 4 (2005), 275–291. Member-Cormac Flanagan.

[26] Goetz, B. Java theory and practice: More flexible, scalable locking in jdk 5.0, October 2004. Available at `http://www.ibm.com/developerworks/java/library/j-jtp10264/`.

[27] Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., and Lea, D. *Java Concurrency in Practice.* Addison-Wesley Professional, May, 2006.

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

[29] Haller, P., and Odersky, M. Event-based programming without inversion of control. In *Proc. Joint Modular Languages Conference* (2006), Springer LNCS.

[30] Harris, T. Exceptions and side-effects in atomic blocks. In *In PODC Workshop on Concurrency and Synchronization in Java Programs.* (2004).

[31] Harris, T., and Fraser, K. Language support for lightweight transactions. In *OOPSLA'03* (2003), pp. 388–402.

[32] 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).

[33] Herlihy, M., Luchangco, V., Moir, M., and William N. Scherer, I. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing* (Jul 2003), pp. 92–101.

[34] Herlihy, M., and Moss, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (May 1993), pp. 289–300.

[35] 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).

[36] Hoare, C. A. R. *Towards a theory of parallel programming.* Springer-Verlag New York, Inc., New York, NY, USA, 2002, pp. 231–244.

[37] Holzmann, G. J. The model checker SPIN. *Software Engineering 23*, 5 (1997), 279–295.

[38] javagrande. The java grande forum benchmark suite. Available at `www.epcc.ed.ac.uk/javagrande`.

[39] Konstantin, K., and Artho, C. Jlint. http://jlint.sourceforge.net/.

[40] Kung, H. T., and Robinson, J. T. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 6*, 2 (1981), 213–226.

[41] Lamport, L., and Schneider, F. B. Pretending atomicity. Tech. Rep. TR89-1005, Digital Equipment Corporation, 1989.

[42] Lea, D. *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley, 2000.

[43] Leijen, D., and Hall, J. Parallel performance: Optimize managed code for multi-core machines, October 2007. Available at `http://msdn.microsoft.com/msdnmag/issues/07/10/Futures/default.aspx`.

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

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

[46] Liu, Y. D., and Smith, S. F. Expressive Ownership with Pedigree Types (long version), `http://www.cs.jhu.edu/~yliu/pedigree`. Tech. rep., The Johns Hopkins University, Baltimore, Maryland, March 2007.

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

[48] 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.

[49] 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.

[50] 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.

[51] Miller, M. The E Language, `http://www.erights.org`.

[52] Moss, J. E. B., and Hosking, A. L. Nested transactional memory: model and architecture sketches. *Sci. Comput. Program. 63*, 2 (2006), 186–201.

[53] 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).

[54] Noble, J., Potter, J., and Vitek, J. Flexible alias protection. In *ECOOP'98* (Brussels, Belgium, July 1998).

[55] 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.

[56] Onodera, T., Kawachiya, K., and Koseki, A. Lock reservation for java reconsidered. In *ECOOP 2004* (2004).

[57] Ringenburg, M. F., and Grossman, D. AtomCaml: First-class atomicity via rollback. In *ICFP'05* (September 2005), pp. 92–104.

[58] Roscoe, A. *The Theory and Pracitce of Concurrency.* Prentice Hall, Boston,Massachusetts, 1997. http://web.comlab.ox.ac.uk/oucl/work/bill.roscoe/publications/68b.pdf.

[59] Shavit, N., and Touitou, D. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing* (Aug 1995), pp. 204–213.

[60] Spiegel, A. Pangaea: An automatic distribution front-end for java. In *Fourth IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '99)* (1999).

[61] 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.

[62] von Praun, C. Detecting synchronization defects in multi-threaded object-oriented programs, 2004. PhD thesis.

[63] Welc, A., Hosking, A. L., and Jagannathan, S. Transparently reconciling transactions with locking for java synchronization. In *ECOOP'06* (2006), pp. 148–173.

[64] 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.

[65] Williams, A., Thies, W., and Ernst, M. D. Static deadlock detection for java libraries. In *ECOOP 2005* (2005).

[66] Yonezawa, A., Briot, J.-P., and Shibayama, E. Object-oriented concurrent programming abcl/1. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications* (New York, NY, USA, 1986), ACM Press, pp. 258–268.

Vita

Xiaoqi Lu was born in Chongqing, China on June 7, 1975. After receiving the degree of Bachelor of Science in Information Science and Technology from Beijing Normal University in 1997, she entered the Master's program in the same institute as the top one student of her class. She earned the degree of Master of Science in 2000. Since then, she has been studying for her Ph.D. at The Johns Hopkins University. Her research interests include concurrency programming, language security, virtual machine architecture, and distributed systems.