The Dissertation Committee for David Wilson Kitchin
certifies that this is the approved version of the following dissertation:

Orchestration and Atomicity

Committee:

Jayadev Misra, Supervisor

William Cook, Supervisor

Don Batory

Keshav Pingali

Dan Grossman

Orchestration and Atomicity

by

David Wilson Kitchin, B.S.C.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2013

This dissertation is dedicated to my mother and father,
whose boundless love and support have made all of this possible.

Acknowledgments

First, I must express my gratitude to my best friend, Kristine Butler, without whose unwavering support I could never have completed this journey.

I am also deeply grateful to my good friends Chris Lundberg and Mercedes Vaughn, who have given me a place to call home for the past three years, and whose wonderful companionship I have continually enjoyed.

I am thankful every day for my loving, quirky, and brilliant family.

I am grateful for my excellent advisor and mentor, Jay Misra, so much so that I am not sure how to put it into words. We have worked together for eight years, and our collaborations have always been productive and enjoyable. He has always treated me with respect, even when I have ignored his guidance or strained his patience. Jay is a great scholar, and I look forward to opportunities to work with him again in the future.

My gratitude extends to all of the members of the Orc Research Group, past and present, who have each in their own way made the group a lively and interesting hub of great ideas and vigorous discussion. My current co-conspirators, John Thywissen and Arthur Peters, are at the top of that list.

I would like to thank the other four members of my dissertation committee — William Cook, Don Batory, Keshav Pingali, and Dan Grossman — for their patience throughout my long and difficult writing process.

I would like to thank all of the organizers, lecturers, and participants who

attended the Summer School on Language-Based Techniques for Concurrent and Distributed Software at the University of Oregon, in 2006. That confluence of great ideas and talented minds gave me the initial spark that led, over the course of seven years, to the development of Ora.

I am indebted to Frank Pfenning, who managed to teach me enough about programming language theory in a single semester to last me through a decade of exploration and research, with more still to come.

I am also indebted to Steven Rudich, who managed to teach me enough about the magic of computer science in a single semester to last me through a dozen years of programming and proving, with more still to come.

And I am indebted to a whole host of other excellent teachers for their innumerable lessons: Stephen Gregory, Meg Curran, Richard Monroe, Paul Jourcin, Mark Stehlik, Klaus Sutner, Patricia Carpenter, and so many others.

Lastly, I would like to thank Zoë Keating for her beautiful music, which has helped me to endure a long journey and to remain an optimist throughout.

David Wilson Kitchin

The University of Texas at Austin
August 2013

Orchestration and Atomicity

Publication No. ____________________

David Wilson Kitchin, Ph.D.
The University of Texas at Austin, 2013

Supervisor: Jayadev Misra
Co-supervisor: William Cook

This dissertation presents the concurrent programming language Ora, an extension of the Orc orchestration language with the capability to execute transactions. A new formal definition of transactions is given, in terms of two complementary properties: atomicity and coatomicity. These properties are described in terms of a partial order of events, rather than as properties of a totally ordered program trace. Atomicity and coatomicity are ensured in Ora programs by a novel algorithm for multiversion concurrency control.

Contents

Chapter 1

Introduction

Writing clear and correct concurrent code is a difficult task, even for experienced programmers. It requires attention to the potential interactions of many different entities, which are often in disparate locations in a program, or even in separate programs entirely. Moreover, most modern programming languages do not provide convenient or well-integrated concurrency primitives. Their fundamental designs were established in an era predating widespread concurrent computing. What if we started with a language specifically designed for concurrent programming?

1.1 Controlling Structured Concurrency

The Orc programming language provides *structured concurrency*, a form of concurrency where the program structure is directly related to the execution structure, making it easier to modularize and reason about concurrent activity. Chapter 2 presents the Orc calculus, a simple process calculus for structured concurrency, and then shows how that calculus can be expanded into a full programming language. A full formal semantics of Orc is also presented, including some features that have not previously been described formally, such as the 'otherwise' combinator, and sites that manipulate mutable state.

Unfortunately, while Orc makes concurrency easier to express, it does not make it easier to control. In particular, concurrent access to shared memory, which has always been a problem for concurrent programs, is particularly difficult to control in Orc. So, in search of a suitable method for Orc to use, Chapter 3 surveys methods of concurrency control in three broad categories: mutual exclusion using locks, message passing, and optimistic concurrency using transactional memory.

Locks

Locks were the earliest method of concurrency control. They can ensure that only one process at a time is running certain pieces of code or accessing certain parts of shared memory. This allows us to reuse single-process reasoning techniques on concurrent code. Single-process reasoning is already of questionable utility for a pervasively concurrent language, and even as locks solve some problems, they introduce others, well-known to concurrent programmers, such as deadlock, loss of modularity, and difficulty in formal reasoning.

Messages

Some languages avoid the problems of concurrent access to shared memory by restricting the interaction of processes to the sending and receiving of messages. While this can be very beneficial, it often requires a significant change in programming style. Furthermore, existing message-passing languages tend to express *unstructured concurrency*; many important properties of the system emerge from the interaction between processes, and so they are difficult to understand by looking at individual process definitions. So, this model is not a great match for Orc; while Orc programs are perfectly capable of creating communication channels and passing messages around, restricting them to use only this form of shared state is too confining.

Transactional Memory

Sequences of program actions are organized into transactions, which are run in such a way that they appear to have happened either all at once, or not at all. Furthermore, transactions are allowed to run concurrently with each other, even if it is possible that they will interfere with each other. The changes a transaction makes to shared memory are hidden while it runs, so that if interference does occur, the transaction can be aborted, and all of its changes undone. If the transaction completes without interference, it can commit, and its changes become visible all at once.

Despite being friendlier to concurrent execution, transactions are still not designed for pervasive concurrency. In particular, concurrent activity within a transaction is rarely allowed. Furthermore, the theory for reasoning about transactions, as with locks, is based on reasoning about ordered sequences of events called traces that are arbitrarily scrambled, and then must be shown to be equivalent to an unscrambled sequence of events. This is a poor formal model for Orc. However, if we approach transactions from a new perspective, and define them using a different model of concurrent execution, a much more effective form of concurrency control arises.

1.2 Contributions of This Thesis

This thesis makes multiple contributions to concurrent programming language design and the theory of transactional memory: a new principle of concurrency control, an extended version of Orc with support for transactions, a novel algorithm for versioned transactional memory, and formal definitions for all of these contributions.

A New Principle of Concurrency Control

How might we better describe and implement transactions for a pervasively concurrent language? Chapter 4 describes how, with a simple change of perspective, we can obtain an intuitive and general principle of concurrency control that corresponds to a new definition for transactions. This novel principle is the central contribution of this thesis.

Rather than starting with traces, and trying to unscramble them, instead we start with a set of events, and then impose a partial order between some of the events based on their causal relationship. We can then take a subset of the events, designate it an *atom*, and ask for two simple properties:

ATOMICITY: Whenever an event in an atom is a cause for some event outside of the atom, every event in the atom is a cause for that outside event.

COATOMICITY: Whenever an event outside of an atom is a cause for an event within the atom, that outside event is a cause for every event within the atom.

These simple, symmetric properties are all that we need to define transactions. There is no need to talk about traces or interleaving; in fact, this principle has nothing to do with sequential execution at all. There is a profound lesson here: *sequentiality is not essential to transactions.*

An Extension of Orc with Transactions

Following the example of other languages that implement transactional memory, we can make this new form of transaction into a language feature. Chapter 5 describes Ora, an extension of Orc with a new combinator called **atomic**. An expression inside **atomic** is run in such a way that it behaves as an atom, as described above: the set of events that happen within the expression must obey atomicity and coatomicity. The **atomic** combinator allows us to easily write concurrent programs with well-controlled shared state.

A New Multiversioning Algorithm

Chapter 6 describes the underlying algorithm that implements atomicity and coatomicity. It is a novel variant of an established technique called multiversion concurrency control. This algorithm maintains two version systems: a logical version system that acts as a distributed snapshot algorithm, keeping track of which events can see which other events, and a separate resource version system that determines whether a transaction can merge its changes while maintaining consistency.

Formal Semantics and Definitions

Unlike many other transactional memory systems, the Ora language and its underlying concurrency principles are formally and precisely defined. Chapter 7 presents a complete formal semantics for the Ora programming language, including its underlying versioning algorithms. Chapter 8 presents formal definitions of the properties of atomicity, coatomicity, and consistency.

Potential Extensions

Chapter 9 discusses some problems with Ora, and some possible future extensions, such as new transactional sites, how to incorporate real and simulated time, and how to program with sites that could be waiting for events to happen in other transactions.

Chapter 2

Orc

Orc is a programming language designed with concurrency in mind. Orc supports *orchestration*, a programming style where activities are concurrent but their organization is hierarchical. The lexical structure of the Orc program is reflected in the structure of the concurrent computation. This is a form of *structured concurrency*, by analogy to the *structured programming* advocated by Dijkstra [Dij68], and in contrast to the relatively unstructured concurrency of message-passing languages such as Erlang [Arm07] or models such as the π-calculus [Mil99].

Orc emphasizes the importance of interactions with the external world. It interacts with external services, called *sites*, to provide a variety of capabilities. In fact, the notion of sites is so pervasive that even primitive features of the language are implemented as sites.

The Orc programming language is based on a simple process calculus called the Orc calculus. We begin with an explanation of this calculus, and then proceed to a description of the full programming language, and then a formal presentation of the calculus with an operational semantics.

2.1 Orc as a Process Calculus

The Orc process calculus is based on the execution of *expressions*. As an expression executes, it may interact with external services (called *sites*), and may *publish* values during its execution. An expression may also *halt*, explicitly indicating that its execution is complete.

Simple expressions are built up into more complex ones by using *combinators* and by defining *functions*.

2.1.1 Values

A value, on its own, is a valid Orc expression. When the expression is executed, it publishes that value, and then halts.
An Orc value could be:

- A number, such as `0`, or `1.618`, or `-7`
- A boolean, `true` or `false`.
- A character string, such as `"ceci n'est pas une |"`
- The `signal` value, which carries no information.

There are many other types of values created and used during the execution of Orc expressions; these are simply the types most likely to be written as expressions on their own.

2.1.2 Sites

Orc interacts with external services through *site calls*. A site call resembles a function call in other languages. It names the site to be called, and gives arguments for the call. The external service then performs some computation, and may eventually *respond* to the call, either with a single value, or the special response `stop`, indicating the absence of a value. If the response is a value, that value is published, and the site call then halts. If the response is `stop`, the site call simply halts. A site responds at most once to a call.

Here are some examples of site calls:

- The `Print` site prints text to the console. `Print("Hello, World.")` prints `Hello, World.` on the console, then publishes `signal`, then halts.
- The `Prompt` site asks the user for text input. `Prompt("username:")` presents a prompt with the text `username:`, receives a line of input, publishes that line of input as a string, and then halts.

- The `(+)` site responds with the sum of its arguments. `(+)(2,3)` publishes `5` and then halts.
- The `Rwait` site waits for a given number of milliseconds and then responds. `Rwait(1000)` waits for one second, then publishes `signal`, then halts.

An Orc expression may make use of many other sites, with capabilities such as file system access, Internet search, execution of code written in other languages, and more.

Sites are first-class in Orc, which means they may be used wherever a value could be used. In particular, they can be passed as arguments to calls, and they can be published.

The Orc calculus does not include certain primitive operations that are usually regarded as essential language features. For example, it does not define arithmetic operations such as addition or multiplication, nor does it define any data structures such as lists, nor any facilities for using mutable state. These capabilities, and many others, are provided through site calls instead.

2.1.3 Stop

The expression **stop**, when executed, halts immediately.

2.1.4 Combinators

Orc has four concurrent combinators, which combine subexpressions according to four distinct patterns of concurrent execution.

Parallel Combinator

The parallel combinator, written $f \mid g$, executes expressions f and g concurrently. They have no direct interaction with each other. Each publication of f or g is published by $f \mid g$. When both f and g have halted, $f \mid g$ halts.

Sequential Combinator

The sequential combinator, written $f > x > g$, executes expression f. The *variable* x may appear in expression g, wherever a value could appear. Whenever f publishes a value v, a new copy of g is executed in parallel with f $>x>$ g. In this copy of g, the variable x is replaced by v.

The sequential combinator may be written without a variable name, as $f \gg g$. This is equivalent to writing f $>x>$ g with an x that never appears in g.

When f and all copies of g have halted, f $>x>$ g halts.

Pruning Combinator

The pruning combinator, written f $<x<$ g, executes expressions f and g concurrently. The variable x may appear in expression f, wherever a value could appear.

When g publishes a value v, execution of g is immediately *terminated*, and all occurrences of x in f are replaced by v. If g halts without publishing a value, all occurrences of x in f are instead replaced by **stop**. A site call with an argument of **stop** halts immediately.

Since f begins executing before x is replaced, execution of f may encounter a site call with an argument of x, or even x as its own expression. Execution of such expressions *blocks*; nothing happens until x is replaced by a value or **stop**, and then the expression resumes execution. A blocked expression is not considered halted, since it might become unblocked in the future.

When f has halted, and g has either published or halted, f $<x<$ g halts.

Otherwise Combinator

The otherwise combinator, written f **;** g, executes expression f. If f halts without publishing any values, then g is executed.

If f publishes, f **;** g halts when f halts. If f halts without publishing, f **;** g halts when g halts.

2.1.5 Functions

An expression g may be preceded by a function definition, written as follows:

$$\begin{array}{l}\textbf{def}\ F(\bar{y}) = f \\ g\end{array}$$

This defines a function F with formal parameters $\bar{y}$ and body f. The definition together with its scope g forms an expression itself, so a function definition may appear in any nested scope, and an expression could be preceded by any number of function definitions in a nested sequence.

When all free variables in f have been replaced by values, then f is considered closed, and a *function closure* is created. All occurrences of F in g are then replaced by this function closure. A function may be recursive, so F could appear in f as well as in g.

If the variable F is encountered during the execution of g, and it has not yet been replaced by a function closure, execution of that use of F blocks, just like in the pruning combinator.

The function closure can be called just like a site in g; function calls are of the form $F(\bar{p})$, where $\bar{p}$ is a sequence of values or variables. However, there is one important difference between site calls and function calls: a function call is *lenient*, meaning that the call may proceed even if some of its parameters $\bar{p}$ have not yet been replaced by values. Some parameters might still be variables, and some might even be **stop**. When the closure is called, the call is replaced by a copy of f in which the formal parameters $\bar{y}$ have been replaced by the actual parameters $\bar{p}$, and any use of F has been replaced by the closure. This copy of f then executes.

Function closures are first-class in Orc, which means they may be used wherever a value could be used. In particular, they can be passed as arguments to calls, and they can be published. A function closure can escape the scope in which it was defined, by being published, or passed to a site. Since a function closure has no free variables, it is self-contained, so this escape is safe.

2.2 Orc as a Programming Language

Orc is not just a process calculus; it is also a full-featured programming language. The calculus is a subset of the full language, so any expression in the calculus is also a valid Orc program. The Orc language has many syntactic constructs that make programming easier and more convenient. However, each of these constructs translates back into the calculus. Thus, each Orc program is also transformable to a valid Orc calculus expression. Some of these translations are straightforward; others are more complex.

The current implementation of Orc has several features that are not discussed here: a sophisticated static typechecker, support for classes, a large site library, and more. For more information, consult the Orc reference manual, which provides extensive documentation.[Orc13a]

2.2.1 `val`

The simplest syntax extension is the **`val`** declaration. It is an alternate form of the pruning combinator, in prefix instead of postfix form. It makes many programs easier to read.

For example, these two programs are equivalent:

```
val x = g
f
```

```
f <x< g
```

2.2.2 Operators

The Orc language has a standard suite of primitive operators:

- Arithmetic operators: `+`, `-`, `*`, `/`, `\%` (modulus), and `**` (exponent)
- Comparators: `<:`, `<=`, `=`, `>=`, `:>`, and `/=`
- Logical operators: `\&\&`, `||`, and `\textasciitilde` (negation)

Each use of these operators is actually translated to a site call. The corresponding sites are named `(+)`, `(-)`, and so forth; they can be used explicitly in an Orc program, though this is very rare.

For example, these two programs are equivalent:

```
1 + 0.618
```

```
(+)(1, 0.618)
```

2.2.3 Conditionals

The Orc language provides a conditional expression of the form

if f **then** g **else** h

where f, g, and h are expressions. Execution of a conditional executes f until it publishes a value, and then terminates it. If that value is `true`, g is executed. If that value is `false`, h is executed. If the value is not a boolean, or if f halts without publishing, then the conditional halts.

This is translated to the Orc calculus using the sites `Ift` and `Iff`. `Ift` responds with **`signal`** if its argument is `true`, otherwise it responds with **`stop`**. `Iff` tests for `false` in the same way.

For example, the following two programs are equivalent:

```
if (x < 0)
  then "negative"
  else "nonnegative"
```

```
(
    Ift(b) >> "negative"
  | Iff(b) >> "nonnegative"
)
    <b< (x < 0)
```

2.2.4 Flattening

The Orc calculus does not allow expressions to appear as arguments to calls. However, in many circumstances, it is convenient to have this capability. So, whenever an expression appears as an argument, that expression is replaced by a fresh variable name (call it x), and then a pruning combinator is added to bind the first publication of the expression to x. The transformation is applied recursively throughout the program until no expression appears as an argument to a call.

For example, these three programs are all equivalent:

```
(4 * 3) - (2 + 1)
```

```
val x = 4 * 3
val y = 2 + 1
x - y
```

```
(-)(x,y)
  <x< (*)(4,3)
  <y< (+)(2,1)
```

This transformation is fundamental to the Orc programming language. It provides ‘implicit’ concurrency. Due to the use of pruning combinators, each nested expression is executed in parallel. In the case of function calls, this capability can be very powerful; since function calls are lenient, the execution of the function body can proceed in parallel with the execution of its arguments.

This capability also extends to all syntactic constructs that are converted to site calls.

2.2.5 Structured Data

The Orc language provides three primitive data structures: tuples, lists, and records.

Tuples are sequences of a fixed length, with at least two elements. They are written like this: `(0, "zero")`. Different elements of a tuple may be different types of values.

Lists are sequences of variable length, with any number of elements. The empty list is written `[]`. Lists with elements are written like this: `[0, 0, 6]`. Typically, all elements of a list are the same type of value. A list may also be formed using the infix *cons* operator, written as a colon (:). It creates a new list from a given head element and tail list. Elements of a list are typically all of the same type, though this is not required.

Records are mappings from names to values. A record is written like this: `{. x = 3, y = 4 .}` . This record has two fields, with labels `x` and `y` mapped to values `3` and `4` respectively. Record elements are accessed using the `.` operator, called ”dot”, like this: `{. z = false .} >r> r.z` .

These data structures are built using sites. Tuples are formed using the `Tuple` site. Lists are formed by a chain of `Cons` site calls ending with a `Nil` call to create the empty list. Records are formed using the `Record` site.

The following two programs are equivalent:

```
  (1, 1)
| [2, 3]
| {. x = 5, y = 8 .}
```

```
  Tuple(1, 1)
| Cons(2, y) <y< Cons(3, x) <x< Nil()
| Record("x", 5, "y", 8)
```

2.2.6 Patterns

In addition to providing the means to build data structures, Orc also provides a way to examine them: patterns. A pattern may replace the variable name in the sequential combinator, the pruning combinator, or the **`val`** declaration. A pattern has the same form as a data structure, except that the values are instead replaced by variable names, or recursively by other patterns. Thus, tuple patterns, list patterns, and record patterns are all possible.

When a value would be bound to the pattern, the structure of the value is *matched* against the pattern, binding components of the value to the corresponding variables in the pattern.

For example, this program uses a tuple pattern to sum the components of some pairs:

```
( (0, 3) | (2, 5) | (1, 7) ) >(x, y)> x+y
```

It is possible for a pattern match to fail. If this happens in a sequential combinator, the published value is simply thrown away; no copy of the right hand expression is executed. If it happens in a pruning combinator, the value is thrown away, and the right hand expression is *not* terminated.

Every expression using a pattern can be converted to an equivalent Orc calculus expression, using a number of helper sites that extract the components of a data structure. However, this conversion is complex; it is beyond the scope of this document.

2.2.7 Join

It is often useful to wait for some number of expressions to each publish a value, and then publish a **`signal`** when all of them have done so. The infix operator `&`, called the "join" operator, provides this capability. Its translation is simple: each expression in the join is made into an element in a tuple; when the tuple is published, it is discarded and **`signal`** is published instead.

The following two programs are equivalent:

```
Rwait(10) & Rwait(100) & Rwait(1000)
```

```
(x,y,z) >> signal
  <x< Rwait(10)
  <y< Rwait(100)
  <z< Rwait(1000)
```

Note that each expression is terminated after it publishes. The join operator is intended to be used primarily with expressions which publish as the last step of their execution, so no computation remains.

2.2.8 Enhanced Function Definitions

Defined functions have an expanded set of capabilities in the Orc language. Any contiguous group of function definitions may be mutually recursive. Furthermore, functions may be defined by using *clauses*: multiple definitions of the same function name, with patterns as arguments. When such a function is called, the arguments are matched against the patterns of the first clause. If any argument matched against a pattern is unbound, the call blocks until that argument is bound. If all patterns successfully match, the body of that clause is executed, with the appropriate bindings. If any pattern fails to match, then the next clause is tried. If no clauses remain, the call halts.

Here is an example of a clausal function, which sums a list of numbers. The style of programming will look familiar to users of typed functional programming languages, such as ML or Haskell.

```
def sum([]) = 0
def sum(h:t) = h + sum(t)
```

Mutually recursive functions and clausal functions can be converted to equivalents in the Orc calculus. However, as with patterns, the details of the conversion are beyond the scope of this document.

2.2.9 Mutable State

In addition to the various language features mentioned so far, Orc programs also rely on a library of sites, providing various capabilities. Orc variables and data structures are immutable, so we must rely on sites to create shared mutable resources. Here we will focus on four essential sites: `Ref`, `Cell`, `Channel`, and `Semaphore`.

Ref

The `Ref` site creates mutable references. It may be invoked with one argument, or no arguments. `Ref(`v`)` creates a new reference with initial value v. `Ref()` creates a new reference with no initial value; it is *empty*. In either case, the response is a record r, with two fields: `read` and `write`. Each field is a site, which reads or writes the reference, respectively.

- r`.read()` responds with the current value of the reference. If the reference is empty, then the call blocks until a write occurs.

- r`.write(`v`)` sets the value of the reference to v, and responds with **`signal`** . If the reference was empty, each read currently blocking is unblocked and responds with v.

For syntactic convenience, r`?` is equivalent to r`.read()`, and r `:=` v is equivalent to r`.write(`v`)`.

Cell

The `Cell` site creates write-once mutable cells. It takes no arguments. `Cell()` creates a new empty cell, and responds with a record r, with two fields: `read` and `write`. Each field is a site, which reads or writes the cell, respectively.

- r`.read()` responds with the current value of the cell if it has been written, or blocks if it is still empty.
- If the cell is empty, r`.write(`v`)` sets the value of the cell to v and responds with **`signal`**. This unblocks all readers.
- If the cell has a value, r`.write(`v`)` does nothing, and halts.

For syntactic convenience, r`?` is equivalent to r`.read()`, and r `:=` v is equivalent to r`.write(`v`)`.

Channel

The `Channel` site creates asynchronous FIFO buffers. It takes no arguments. `Channel()` creates a new buffer with no contents, and responds with a record r, with two fields: `get` and `put`. Each field is a site, which takes from or adds to the buffer, respectively.

- r`.put(`v`)` adds v to the end of the buffer, and responds with **`signal`**
- r`.get()` takes the value v at the front of the buffer, and responds with v. If the buffer is empty, the call waits until a new item is put, and takes that value.

The buffer is fair: if new values are put into the buffer infinitely often, then no get operation will block forever.

Semaphore

The `Semaphore` site creates semaphores. It takes one argument. `Semaphore(`n`)` creates a new semamphore with initial value n (where $n \geq 0$), and responds with a record r, with two fields: `acquire` and `release`. Each field is a site, which decrements ('P') or increments ('V') the semaphore, respectively.

- r`.acquire()` decrements the semaphore value if it is greater than 0, responding with **`signal`**. If the semaphore value is 0, the call blocks.
- r`.release()` increments the semaphore value (possibly unblocking an acquire call) and responds with **`signal`**.

The semaphore is fair: if the semaphore is released infinitely often, then no acquire operation will block forever.

2.3 Formal Semantics

This section presents a formal semantics for the Orc calculus, in four parts. The formal grammar describes the structure of Orc expressions. The *expression semantics* of Orc shows the labeled small-step transition relation on Orc expressions, and the structure of the transition labels. The *environment semantics* of Orc describes the structure of the environment in which an Orc program runs, and the small-step transitions which the program and its environment jointly make as they interact. The environment semantics relies on another relation, $\hookrightarrow$, which processes site computations. The *site semantics* describes how sites compute by defining a set of rules for the $\hookrightarrow$ relation.

Though the operational semantics of Orc has been presented previously[KCM06], the semantic rules given here are more detailed than those seen in previous versions. The internal semantics includes rules for the otherwise combinator, along with an inductive judgment for halting of expressions. Semantic rules for the external environment, and for stateful sites in particular, have never been published before. Each of these inclusions is important and necessary to support the extensions to Orc shown in later chapters.

Since a program in the Orc programming language can be reduced to an equivalent expression in the calculus, it suffices as a formal representation for the full language, assuming that each site used in the program also has some formal representation. The site semantics presented here includes all of the sites needed for the translations mentioned in section 2.2, and it also models the shared resources created by `Ref`, `Cell`, `Channel`, and `Semaphore`, but it does not include any other sites. In particular, `Rwait` is not formally modeled; a formal semantics of Orc with real time is beyond the scope of this document.

2.3.1 Syntax

The formal grammar of the Orc calculus is given here. A grammar for the full Orc programming language can be found in the Orc reference manual.[Orc13b]

$$\begin{array}{rcl}
x, y & \in & \mathit{Variable} \\
k & \in & \mathit{Handle} \\
V & \in & \mathit{Value}
\end{array}$$

$$\begin{array}{rclcl}
v & \in & \mathit{Orc\ value} & ::= & V \mid D \\
w & \in & \mathit{Response} & ::= & v \mid \mathbf{stop} \\
p & \in & \mathit{Parameter} & ::= & w \mid x
\end{array}$$

$$\begin{array}{rclcl}
D & \in & \mathit{Definition} & ::= & \mathbf{def}\ y(\bar{x}) = f
\end{array}$$

$$\begin{array}{rclcll}
f, g & \in & \mathit{Expression} & ::= & p & \text{(Parameter)} \\
 & & & \mid & p(\bar{p}) & \text{(Site or Function Call)} \\
 & & & \mid & k? & \text{(Site Call In Progress)} \\
 & & & \mid & f > x > g & \text{(Sequential Combinator)} \\
 & & & \mid & f \mid g & \text{(Parallel Combinator)} \\
 & & & \mid & f < x < g & \text{(Pruning Combinator)} \\
 & & & \mid & f \ ;\ g & \text{(Otherwise Combinator)} \\
 & & & \mid & D \# f & \text{(Function Definition)}
\end{array}$$

Variables and handles are arbitrary identifiers. The set of values V contains all sites, all constants, all structured values, and all other datatypes returned by sites.

A value can be either an external value as defined by V, or a closure D created within the program. A response w could be any of these, or **stop**, indicating the absence of a value (due to a halted computation). A parameter p could be any of these, or a variable name.

An expression f could be a bare parameter, a site call, a pending site response, any of the four Orc combinators, or a definition scoped to an expression.

2.3.2 Expression Semantics

The semantics of Orc expressions is a small-step operational semantics with labeled transitions.

Transition Labels

Here is the grammar for the labels attached to each transition.

$$\begin{array}{lllll}
l & \in & \textit{Non-publication Label} & ::= & V_k(\bar{v}) & \text{(Call)} \\
 & & & \mid & k?w & \text{(Response)} \\
 & & & \mid & \tau & \text{(Internal Event)} \\
 & & & & & \\
L & \in & \textit{Label} & ::= & !v & \text{(Publication)} \\
 & & & \mid & l &
\end{array}$$

A non-publication label could be a site call, a site response, or an internal event. A label can be any of these, or a publication.

Halted Expressions

The helper judgment f halted indicates when the expression f has halted. A halted expression has no transitions, and will never have any future transitions.

$$\frac{f \text{ halted}}{D \# f \text{ halted}} \qquad (\textsc{HaltDef})$$

$$\frac{f \text{ halted} \qquad g \text{ halted}}{f \mid g \text{ halted}} \tag{HaltPar}$$

$$\frac{f \text{ halted}}{f > x > g \text{ halted}} \tag{HaltSeq}$$

$$\textbf{stop}(\bar{p}) \text{ halted} \tag{HaltCall}$$

$$V(..., \textbf{stop}, ...) \text{ halted} \tag{HaltArg}$$

$$\textbf{stop} \text{ halted} \tag{HaltStop}$$

Orc Transition Rules

Here are the labeled transition rules for Orc expressions.

$$\frac{k \text{ fresh}}{V(\bar{v}) \xrightarrow{V_k(\bar{v})} k?} \tag{SiteCall}$$

$$k? \xrightarrow{k?w} w \tag{SiteReturn}$$

$$v \xrightarrow{!v} \textbf{stop} \tag{Publish}$$

These are the transitions associated with base expressions. Notice that the response w is unbound in the (SiteReturn) rule. This means that an expression $k?$

has an unbounded number of possible transitions at all times, one for each possible response w. However, as we will see later, the expression does not make these transitions autonomously; it must synchronize with an available response in the environment.

$$\frac{f \xrightarrow{L} f'}{D \# f \xrightarrow{L} D \# f'} \qquad (\textsc{DefScope})$$

$$\frac{\begin{array}{c} D \text{ is } \mathbf{def}\ y(\bar{x}) = g \\ FV(D) = \emptyset \end{array}}{D \# f \xrightarrow{\tau} [y \mapsto D]f} \qquad (\textsc{DefClose})$$

$$\frac{D \text{ is } \mathbf{def}\ y(\bar{x}) = g}{D(\bar{p}) \xrightarrow{\tau} [y \mapsto D][\bar{x} \mapsto \bar{p}]g} \qquad (\textsc{DefCall})$$

These are the transitions used to manage function definitions and function closures.

The notation $FV(D)$ used in (DefClose) means "the set of free variables of D". Thus, a closure is not considered closed until its set of free variables is empty. This will happen when each free variable in D has been replaced by a value.

These rules are more comprehensive than in previous presentations of the Orc semantics. In particular, note that (DefCall) is as powerful as the β-reduction rule of the λ-calculus. Indeed, using this semantics, Orc embeds the untyped λ-calculus.

The remaining transition rules, governing the Orc combinators, are shown in Figure 2.1.

2.3.3 Environment Semantics

In addition to the transitions of the Orc program, there is another set of semantic rules describing the environment in which the program runs, and the interactions between the Orc program and that environment.

$$\frac{f \xrightarrow{L} f'}{f \mid g \xrightarrow{L} f' \mid g} \quad (\textsc{ParL})$$

$$\frac{f \xrightarrow{l} f'}{f >x> g \xrightarrow{l} f' >x> g} \quad (\textsc{SeqN})$$

$$\frac{g \xrightarrow{L} g'}{f \mid g \xrightarrow{L} f \mid g'} \quad (\textsc{ParR})$$

$$\frac{f \xrightarrow{!v} f'}{f >x> g \xrightarrow{\tau} f' >x> g \mid [x \mapsto v]g} \quad (\textsc{SeqV})$$

$$\frac{f \xrightarrow{l} f'}{f \; ; \; g \xrightarrow{l} f' \; ; \; g} \quad (\textsc{OtherN})$$

$$\frac{f \xrightarrow{!v} f'}{f \; ; \; g \xrightarrow{!v} f'} \quad (\textsc{OtherV})$$

$$\frac{f \text{ halted}}{f \; ; \; g \xrightarrow{\tau} g} \quad (\textsc{OtherH})$$

$$\frac{f \xrightarrow{L} f'}{f <x< g \xrightarrow{L} f' <x< g} \quad (\textsc{PruneL})$$

$$\frac{g \xrightarrow{l} g'}{f <x< g \xrightarrow{l} f <x< g'} \quad (\textsc{PruneN})$$

$$\frac{g \xrightarrow{!v} g'}{f <x< g \xrightarrow{\tau} [x \mapsto v]f} \quad (\textsc{PruneV})$$

$$\frac{g \text{ halted}}{f <x< g \xrightarrow{\tau} [x \mapsto \textbf{stop}]f} \quad (\textsc{PruneZ})$$

Figure 2.1: Semantics of the Orc Combinators

Environment and Event Grammar

An environment, denoted by E, is a set of events, with the following grammar:

$$\begin{array}{rcl} e \in \mathit{Events} & ::= & \langle k \triangleright V(\bar{v}) \rangle \\ & \mid & \langle k \triangleleft w \rangle \\ & \mid & ... \end{array}$$

A site call event $\langle k \triangleright V(\bar{v}) \rangle$ indicates that the site V was called with arguments $\bar{v}$ and handle k.

A site return event $\langle k \triangleleft w \rangle$ indicates that the site call associated with handle k has finished computing, and that the response was w.

Each site will also extend the grammar of events with new events representing its own operations.

External Transitions

An Orc program f executing within an environment E is denoted by the pair E, f. The following small-step semantic rules specify the transition relation $\longrightarrow$ on such a pair. Each such step involves a transition within the program f and some corresponding activity in the environment E.

$$\frac{f \xrightarrow{\tau} f'}{E, f \longrightarrow E, f'} \qquad (\textsc{EnvTau})$$

$$\frac{f \xrightarrow{!v} f'}{E, f \longrightarrow E, f'} \qquad (\textsc{EnvPub})$$

$$\frac{f \xrightarrow{V_k(\bar{v})} f'}{E, f \longrightarrow E \cup \{\langle k \triangleright V(\bar{v}) \rangle\}, f'} \qquad (\textsc{EnvCall})$$

$$\frac{\begin{array}{c}\langle k \triangleright V(\bar{v})\rangle \in E \\ \langle k \triangleleft \_\rangle \notin E \\ E \vdash V(\bar{v}) \overset{F}{\hookrightarrow}\ w\end{array}}{E, f \ \longrightarrow\ E \cup F + \langle k \triangleleft w\rangle, f'} \qquad (\textsc{EnvProcess})$$

$$\frac{\begin{array}{c}\langle k \triangleleft w\rangle \in E \\ f \ \xrightarrow{k?w}\ f'\end{array}}{E, f \ \longrightarrow\ E, f'} \qquad (\textsc{EnvRespond})$$

- The (EnvTau) rule allows a τ transition in the program, producing no change in the environment.

- The (EnvPub) rule similarly allows a toplevel publication from the program, producing no change in the environment. In an actual implementation of Orc, such a publication might be reported on the console, but here it has no real effect on the environment state.

- The (EnvCall) rule allows the program to emit a call, adding a corresponding call event to the environment.

- The (EnvProcess) rule takes a call in the environment and processes it, using the relation $\hookrightarrow$, which defines the behavior of sites. The response to the call, w, is then added to the environment as a new site response event, along with the set of side effects F. Note that a site call computation may only proceed if no response exists yet; this prevents spurious reruns of the same call.

- The (EnvRespond) rule matches a site response with an available response action in the program, so that the response from the site call appears in the new program. The handle k is eliminated from the program by the $k?w$ transition, so the site call can receive at most one response.

The transition relation never removes or modifies events in the environment; it only adds new ones. Thus, all transitions are monotonic in the environment.

2.3.4 Site Semantics

Sites have their own semantic rules, each of which defines the behavior of the $\hookrightarrow$ relation used in the (EnvProcess) rule of the environment semantics when a call to that site is processed. Some sites also create new types of events in the environment in order to track the state of a shared resource.

Events

Sites that create and modify shared mutable resources must keep track of the state of those resources in the environment. Since the environment is designed to be monotonic – that is, events may only be added, never removed or modified – each event represents a new version of a resource's state, and the versions are ordered by a nonnegative sequence number, n. Whenever a resource's state is needed, the site will look for the event with the maximum sequence number to discover the current resource state.

Different instances of the same resource type, e.g. different memory locations or channel buffers, must also be kept separate. Each event is associated with a particular resource identifier R, and when a site is created, its operations are defined only for that particular resource R.

Here is the grammar for the new events, and their components:

$$
\begin{array}{rcll}
e & \in & \mathit{Events} & ::= \; \ldots \\
 & & & \;\;\;\; \mid \;\; \langle \textsc{Ref}_R(\, \boxed{v} \,), n \rangle \\
 & & & \;\;\;\; \mid \;\; \langle \textsc{Channel}_R(\bar{v}), n \rangle \\
 & & & \;\;\;\; \mid \;\; \langle \textsc{Semaphore}_R(n), n \rangle \\
\boxed{v} & \in & \mathit{Contents} & ::= \; v \mid \square \\
R & \in & \mathit{Resource} & \\
n & \in & \mathbb{N} &
\end{array}
$$

Pure Sites

A *pure* site behaves identically regardless of the environment in which it is called, and it adds no new events to the environment. A pure site need not be total; it may respond with `stop` to some calls.

Most of the sites underlying Orc language features are pure. All of the arithmetic operators, logical operators, comparators, data structuring sites, and pattern matching sites are pure. The sites `Ift` and `Iff` are also pure.

All pure sites are governed by a single semantic rule:

$$\frac{V(\bar{v}) = w}{E \vdash V(\bar{v}) \overset{\emptyset}{\hookrightarrow} w} \qquad (\textsc{PureSite})$$

The premise $V(\bar{v}) = w$ calculates the relationship between the input $\bar{v}$ and the response w of the pure site V. This premise will be undefined if the site is not pure. For sake of brevity, the full definitions of each pure site are not listed here; they are obvious from their definitions. The most important characteristic of this rule is that the site computation ignores E entirely, and produces an empty set of side effects. This is what characterizes pure sites.

Ref

The `Ref` site creates mutable references. The site itself, and the sites it creates, are governed by the following rules:

$$\frac{\begin{array}{c} R \text{ fresh in } E \\ e = \langle \textsc{Ref}_R(\Box), 0 \rangle \\ v = \{.\ \ \texttt{read} = \mathit{Read}_R\texttt{,}\ \ \texttt{write} = \mathit{Write}_R\ \ .\} \end{array}}{E \vdash \texttt{Ref}() \overset{\{e\}}{\hookrightarrow} v} \qquad (\textsc{RefNSite})$$

$$\frac{\begin{array}{c} R \text{ fresh in } E \\ e = \langle \textsc{Ref}_R(u), 0 \rangle \\ v = \{.\ \ \texttt{read} = \mathit{Read}_R\texttt{,}\ \ \texttt{write} = \mathit{Write}_R\ \ .\} \end{array}}{E \vdash \texttt{Ref}(u) \overset{\{e\}}{\hookrightarrow} v} \qquad (\textsc{RefVSite})$$

$$\frac{\max_R(E) = \langle \textsc{Ref}_R(v), n \rangle}{E \vdash \mathit{Read}_R() \overset{\emptyset}{\hookrightarrow} v} \qquad (\textsc{ReadSite})$$

$$\frac{\begin{array}{c} \max_R(E) = \langle \textsc{Ref}_R(\_), n \rangle \\ e = \langle \textsc{Ref}_R(v), n+1 \rangle \end{array}}{E \vdash \mathit{Write}_R(v) \overset{\{e\}}{\hookrightarrow} \texttt{signal}} \qquad (\textsc{WriteSite})$$

The (RefNSite) and (RefVSite) rules govern the creation of new references via calls to the `Ref` site. In each case the call creates a new event representing the initial state of the reference, and adds it to the environment as a side effect of the call.

The return value of the call to `Ref` is a record with two fields: `read` and `write` , which are assigned the values Read_R and Write_R, respectively. These methods are the read and write operations of the reference; their behavior is described by the other two rules. This same approach will be used for every site definition that has methods.

The (ReadSite) rule performs a read operation. It locates the reference state for this reference (identified by R) with the largest version number, and returns its contained value v. Note that if the maximum reference state contains $\Box$ (indicating an empty reference) instead of a value, then the (ReadSite) rule does not apply, and so the call blocks since no transition is possible yet. Once the reference has been written, then (ReadSite) will be applicable and the read can proceed.

The (WriteSite) rule performs a write operation. It determines the largest current version number for this reference, creates an event containing the newly written value and a strictly larger version number, and adds this event to the environment as a side effect of the call.

Cell

The `Cell` site creates write-once mutable references. The site itself, and the sites it creates, are governed by the following rules:

$$\frac{\begin{array}{c} R \text{ fresh in } E \\ e = \langle \textsc{Ref}_R(\Box), 0\rangle \\ v = \{\texttt{.}\ \ \texttt{read = } \mathit{Read}_R\texttt{, write = } \mathit{Writeonce}_R\ \texttt{.\}} \end{array}}{E \vdash \texttt{Cell}() \overset{\{e\}}{\hookrightarrow} v} \qquad (\textsc{CellSite})$$

$$\frac{\begin{array}{c} \max_R(E) = \langle \textsc{Ref}_R(\Box), n\rangle \\ e = \langle \textsc{Ref}_R(v), n+1\rangle \end{array}}{E \vdash \mathit{Writeonce}_R(v) \overset{\{e\}}{\hookrightarrow} \texttt{signal}} \qquad (\textsc{WriteOnceN})$$

$$\frac{\max_R(E) = \langle \textsc{Ref}_R(v), \_\rangle}{E \vdash \mathit{Writeonce}_R(\_) \overset{\emptyset}{\hookrightarrow} \texttt{stop}} \qquad (\textsc{WriteOnceV})$$

The (CellSite) rule governs the creation of new cells. A call to `Cell` creates a new event representing the initial state of the cell, and adds it to the environment as a side effect of the call. Note that the state representation of a cell is identical to that of a reference, but the initial state is always □, indicating an empty reference.

The call to `Cell` returns a record with the same read operation as an ordinary reference, but with a different write operation. The (WriteOnceN) and (WriteOnceV) rules capture the behavior of this new write operation. When writing to an empty cell, the behavior is the same as an ordinary reference write. However, writing to a cell that already contains a value causes the call to halt, with no effect.

Channel

The `Channel` site creates asynchronous channels. The site itself, and the sites it creates, are governed by the following rules:

$$\frac{\begin{array}{c} R \text{ fresh in } E \\ e = \langle \textsc{Channel}_R(\epsilon), 0 \rangle \\ v = \{.\ \ \texttt{get} = \mathit{Get}_R,\ \texttt{put} = \mathit{Put}_R\ \ .\} \end{array}}{E \vdash \texttt{Channel}() \overset{\{e\}}{\hookrightarrow} v} \qquad (\textsc{ChannelSite})$$

$$\frac{\begin{array}{c} \max_R(E) = \langle \textsc{Channel}_R(v\,\bar{v}), n \rangle \\ e = \langle \textsc{Channel}_R(\bar{v}), n+1 \rangle \end{array}}{E \vdash \mathit{Get}_R() \overset{\{e\}}{\hookrightarrow} v} \qquad (\textsc{GetSite})$$

$$\frac{\begin{array}{c} \max_R(E) = \langle \textsc{Channel}_R(\bar{v}), n \rangle \\ e = \langle \textsc{Channel}_R(\bar{v}\,v), n+1 \rangle \end{array}}{E \vdash \mathit{Put}_R(v) \overset{\{e\}}{\hookrightarrow} \texttt{signal}} \qquad (\textsc{PutSite})$$

The (ChannelSite) rule governs the creation of new channels. A call to `Channel` creates a new event representing the initial state of the channel, and adds it to the environment as a side effect of the call. A newly created channel is initially empty.

The (GetSite) rule performs a get operation. It locates the state for this channel that has the largest version number, examines the sequence of values it contains, with first element v and remaining elements $\bar{v}$, and then returns the first element v. The call creates a new event representing the modified state of the channel, now containing only the tail $\bar{v}$, makes its version the largest one, and adds it to the environment as a side effect. Note that if the maximum channel state contains an empty sequence, then the (GetSite) rule does not apply, and so the call blocks since no transition is possible yet. Once the channel contains at least one value, then (GetSite) will be applicable.

The (PutSite) rule performs a put operation. It locates the state for this channel that has the largest version number, examines the sequence $\bar{v}$ of values tht it contains, then creates a new event representing the modified state of the channel, now containing $\bar{v}$ with the argument u appended, makes the version of this event the largest one, and adds the event to the environment as a side effect.

Semaphore

The `Semaphore` site creates semaphores. The site itself, and the sites it creates, are governed by the following rules:

$$\frac{\begin{array}{c} R \text{ fresh in } E \\ e = \langle \textsc{Semaphore}_R(i), 0 \rangle \\ v = \texttt{\{.}\ \ \texttt{acquire = } \mathit{Acquire}_R\texttt{,}\ \ \texttt{release = } \mathit{Release}_R\ \ \texttt{.\}} \end{array}}{E \vdash \texttt{Semaphore}(i) \overset{e}{\hookrightarrow}\ v} \quad (\textsc{SemaphoreSite})$$

$$\frac{\begin{array}{c} \max_R(E) = \langle \textsc{Semaphore}_R(i), n \rangle \qquad i > 0 \\ e = \langle \textsc{Semaphore}_R(i-1), n+1 \rangle \end{array}}{E \vdash \mathit{Acquire}_R() \overset{\{e\}}{\hookrightarrow}\ \texttt{signal}} \quad (\textsc{AcquireSite})$$

$$\frac{\begin{array}{l}\max_R(E) = \langle \textsc{Semaphore}_R(i), n\rangle \\ e = \langle \textsc{Semaphore}_R(i+1), n+1\rangle\end{array}}{E \vdash \mathit{Release}_R() \overset{\{e\}}{\hookrightarrow} \texttt{signal}} \qquad (\textsc{ReleaseSite})$$

The (SemaphoreSite) rule governs the creation of new semaphores. A call to `Semaphore` creates a new event representing the initial state of the semaphore, and adds it to the environment as a side effect of the call. A newly created semaphore has the value given by the call's argument.

The (AcquireSite) rule performs an acquire operation. It locates the state for this semaphore that has the largest version number, examines the current value and verifies that it is greater than 0, then creates a new event representing the modified state of the semaphore, now with its count reduced by one, makes the version of this event the largest one, and adds the event to the environment as a side effect. Note that if the maximum semaphore state has a value of 0, then the (AcquireSite) rule does not apply, and so the call blocks since no transition is possible yet. Once the semaphore has a positive value, then (AcquireSite) will be applicable.

The (ReleaseSite) rule performs a put operation. It locates the state for this semaphore that has the largest version number, then creates a new event representing the modified state of the semaphore, with its value increased by one, makes the version of this event the largest one, and adds the event to the environment as a side effect.

Chapter 3

A Survey of Concurrency Control Methods

The greatest strength of the Orc language is also its greatest weakness: by making concurrency pervasive, Orc multiplies all of the problems associated with concurrent access to shared resources. This chapter reviews common strategies for controlling access to shared state in concurrent computations, and considers the applicability of each strategy to Orc programs. We will focus on a particular strategy called transactional memory, which is a good fit for the needs of Orc. The extensions to Orc described in the remainder of this dissertation are inspired by transactional memory.

Throughout, the discussion focuses on how concurrency is controlled, not how it is created or represented. Whether the underlying model uses threads or processes or actors, whether these are created by system calls or associated with certain language objects, and whether they are scheduled cooperatively or preemptively, are all separate considerations. For simplicity, the subsequent descriptions will use the term *process* to refer generically to entities that may execute concurrently.

When concurrent access to shared state is not controlled, many problems can arise. Any process that modifies shared state can affect any other process that can read that shared state. Reasoning about program behavior quickly becomes impossible, as each action of the program could interact with many other interleaved actions originating from disparate parts of the program. A variety of strategies have been developed to mitigate this problem. The most common strategy in current programming practice is to use locks, which ensure that at most one process is performing certain operations by making other processes wait until it is finished. Another strategy is to only allow processes to send messages to each other, and eliminate all other shared state; this rules out many of the problems associated with sharing state, but it requires fundamental changes in programming style. A third strategy is to use transactions, which are sets of operations that must appear to occur all at once, or not at all. A transaction proceeds optimistically, assuming that no concurrent interference will occur; if interference does occur, the operations of the transaction are undone, and it is retried later.

Orc currently has the capability to use two of these strategies, locks and messages, in some form. Adding support for the third strategy, transactions, is one of the main objectives of this dissertation.

3.1 Locks

The dominant approach to concurrency control has been, and largely continues to be, the use of locks. A lock is a shared object with two operations: acquire and release. A lock may be acquired by at most one process at a time; once it has been acquired, it must be released before it can be acquired by a different process. If a process attempts to acquire a lock that is currently held, that process waits, doing nothing, until the lock is released.

The details of lock acquisition depend on the implementation; there are many variations. A process might wait for a lock by repeatedly checking whether the lock is available; this is called a *spin* or a *busy wait*, and it is rarely used due to its inefficiency. More commonly, when a process must wait for a lock, it places itself in the waiting set associated with that lock, from which a subsequent release operation will choose a new holder for the lock. Often, this waiting set is structured as a queue, so that if multiple processes are waiting on a lock, and no process holds that lock forever, then each process waiting on the lock will eventually acquire it. These are called *fair* locks. Certain recursive programs might cause a process to try to acquire a lock it is already holding. Many lock implementations will test for this condition and allow the same process to acquire a lock multiple times; these are called *reentrant* locks.

Since a lock may be held by at most one process at a time, it can be used to protect operations from concurrent interference; we will say that a set of operations is protected by a lock if that lock is acquired before the operations occur, and released when the operations are completed. Whenever that set of operations occurs, no other operations also protected by that lock will be interleaved with those protected operations. If a lock protects a section of the program to prevent it from being executed by multiple concurrent processes, that code is called a *critical section.* If a lock protects every operation that could access a particular piece of shared state, that piece of shared state is called a *monitor.*

Monitors are important enough to receive special treatment in some languages. Object-oriented languages are particularly well suited, since individual objects are easy to designate as monitors. For example, in Java, every object has its own lock, and the `synchronized` keyword may be used to treat any object as a monitor, acquiring the object's lock for the duration of the block or method labeled with `synchronized` [Jav13].

As programs become more complex, multiple locks may be needed, and sometimes a process will need to acquire multiple locks at once. This introduces a host of potential problems [LR80]. The most troublesome of these problems is *deadlock*: two or more processes could wait forever, if each of those processes is waiting to acquire a lock currently held by another of those processes.

Orc has the capability to use locks for concurrency control, by using the `Semaphore` site to create semaphore instances. These locks are fair, since Orc's semaphores use a queue for waiting processes. However, Orc does not provide reentrant locks; since processes have no unique identity, it is impossible to tell if the same process is attempting to acquire a lock multiple times. Orc also does not have a general technique like `synchronized`, since the pieces of shared state it manipulates may not be objects, and are not assured to have a lock associated with them. It is still possible to construct monitors in Orc, but there is no primitive language support for doing so.

Overall, locks are not suitable on their own to control concurrency in Orc. In Orc, pervasive concurrency only exacerbates the complexity of properly acquiring and releasing locks. A disproportionate amount of expertise is required when using locks in Orc, even for simple programs.

3.2 Messages

Many concurrent languages are designed with communication as their fundamental principle. Rather than allowing concurrent processes to arbitrarily share mutable state, these languages allow processes to interact only through a defined messaging system. A process is allowed to send messages to other processes, and receive messages from other processes, but cannot affect other processes in any other way. Interactions within this messaging system are technically operations on shared state, in the sense that the underlying message queues are mutable objects with shared access, but the operations allowed on this state are so much more constrained that they essentially constitute a method of concurrency control in their own right.

There are many examples of programming languages and formal models based on this principle:

- In the Erlang programming language, all processes are separate, self-contained, functional units, which communicate with each other by sending asynchronous messages. A process receives messages in its mailbox, a message queue augmented with pattern matching capabilities [Arm07].

- The π calculus is a formal model of concurrent computation which uses communication channels as primitive objects [Mil99]. The only computation step is the rendezvous of a message sending operation in one process with a message receiving operation in another process. The π calculus allows new channels to be created during an execution, and it also allows channels to themselves be sent as values along other channels. These features give it enough expressive power to embed the λ calculus and express functional programming concepts. The experimental language Pict[1] is a small functional concurrent language which translates down to the π calculus [PT00].

- The join calculus is another formal model of concurrency based on message passing [FG02]. As in the π calculus, messages are sent and received, but in addition, a process may wait on two or more separate messages together before proceeding. The experimental language Polyphonic C# is an extension of C#

[1] The design of Pict, and in particular its translation of language forms into process calculus forms, inspired some of the design of the Orc language.

with the message handling capabilities of the join calculus [BCF04], and its successor Cω also includes these capabilities [Com13].

- The Concurrent ML programming language extends the ML language with new event handling and synchronization primitives that have the same level of expressive power as messages sent over channels [Rep99]. Events in Concurrent ML can be composed to form more complex synchronizations, and the underlying static typing capabilities of ML make these compositions easier to use.

Message passing systems appear to control concurrent state better, by sharing nothing, and thus removing many troublesome concurrent interactions. In particular, messaging systems do not need to use locks to provide mutual exclusion. In many cases, the buffering and ordering of messages suffices to avoid the simple pitfalls. For more complex operations, a particular process can be designated to receive requests for the operation, and then handle those requests one by one, modifying its internal state on each request without the possibility of concurrent interference, and perhaps spawning additional processes. Such a process is called an *actor* [Hew10].

Though message passing systems have these and many other benefits, they require a very different style of programming than conventional shared state programs. Furthermore, message passing programs exhibit *unstructured concurrency*: the behavior of the program emerges from the interactions of separately defined processes, but the structure of these interactions is not reflected in the structure of any individual process; the system must be examined as a whole, taking account of code that may be scattered across disparate process definitions.

Orc can create asynchronous messaging channels using the `Channel` site. Since these channels are first-class values, they can be sent on other channels, giving Orc most of the expressive power of the π calculus.[2] However, Orc is not restricted to use only channels as its form of shared state, so it can mix and match message passing with other forms of shared state. Furthermore, a pure message passing model is not always compatible with Orc programming style, since message passing alone does not take advantage of the structured concurrency offered by Orc combinators.

[2] Orc cannot represent guarded choice, an important component of the π calculus. In Chapter 5, we will see how Ora can represent guarded choice, completing the embedding.

3.3 Transactions

When performing operations on a database, it is useful to aggregate those operations into larger blocks, where the changes made by a block are applied to the database as an indivisible unit. This is called a *transaction.* Transactions have been an object of study in database research and a ubiquitous tool in database practice for decades [GR92].

A transaction is a set of operations that has four properties: Atomicity, Consistency, Isolation, and Durability. The acronym ACID is used to summarize these properties. Atomicity states that the operations are "all or none": either all of the operations are performed on the database, or none of them are. Consistency states that the operations, taken together, must maintain the validity of the database; if the database was valid before the transaction, it remains so after the transaction. Isolation has varying definitions, but generally it requires that the changes made by a transaction not be made visible to other transactions until the transaction has *committed*, meaning that the transaction is finished, its consistency has been verified, and the corresponding changes have been applies to the database. Durability requires that the changes made by the transaction be stored or logged in such a way that they cannot be erased by software errors or machine failure.

In the past two decades, a substantial research effort has arisen to use transactions as a method of concurrency control for program operations that modify shared state. This technique is called *transactional memory.* The concept of transactional memory originally arose as an alternative to locks, in an effort to design *lock-free* data structures [HM93]. The initial design specified a new set of processor instructions that allowed values to be written to memory in such a way that those writes remained hidden until a subsequent *commit* instruction was issued, which would make all of the writes visible at once, providing an "all or none" functionality for memory reads and writes, and thus replacing mutual exclusion techniques such as locks. Subsequent papers proposed *software transactional memory*, or STM, which performed the same function but without requiring hardware support [ST95]. For the purposes of this dissertation, we will restrict our focus to software transactional memory.

In order to execute a transaction in an STM system, the programmer must designate the start and end of the transaction. Early STM systems required special instructions to begin, end, or interrupt a transaction; transactional capabilities were treated as a library, rather than a language feature. More sophisticated systems only require the programmer to mark a block of code marked with a special keyword, such as `atomic` [HF03].

Each memory operation that occurs within a transaction must obey the properties of atomicity, consistency, and isolation.[3] When the block ends, the transaction attempts to commit. If the transaction fails to commit, then it aborts, reverting all of the tentative changes it had made, and retries from the beginning of the block.

As a concurrency control primitive, transactions have a number of advantages. They are not vulnerable to deadlock, or other difficulties associated with lock-enforced mutual exclusion. They move most of the work involved in concurrency control into the implementation layer, leaving the programmer with one simple construct. Programmers are measurably better at using transactions than using fine-grained locking strategies, making fewer errors when writing concurrent programs [RHW10].

These features do come with a price, however. Transactions are still vulnerable to *livelock*: retrying the same actions repeatedly, and thereby failing to make any progress. This can occur when two parallel transactions repeatedly interfere with each other, and consequently they repeatedly abort, and retry. Transactions also impose more (in some cases, substantially more) time and space overhead on many program operations. In this regard, transactional memory has been compared to garbage collection since it trades runtime performance for ease of program reasoning [Gro07].

3.3.1 New Approaches to Transactional Memory

Many languages have incorporated transactions as a language feature and enriched them with new capabilities, or modified the underlying algorithms to improve performance for specific cases. Here are a few examples of the integration of transactions into functional programming languages.

[3] The durability property is not included for transactional memory; the heap is not considered to be durable storage.

AtomCaml

AtomCaml is an extension to OCaml providing a higher-order function `atomic`, which takes a function as an argument and runs it to completion atomically [RG05]. AtomCaml implements `atomic` in an unusual way. A thread executing `atomic` attempts to run the given function to completion, using an eager writing strategy which writes new values to memory directly and maintains an undo log with the previous values. If the thread executes the whole atomic section without being preempted, then it simply discards the undo log and continues; since all of the values have already been written, doing nothing is equivalent to committing. If a context switch occurs before the thread finishes the execution, the previous state of memory is restored using the undo log, to prevent any other thread from seeing the values written by the incomplete transaction; this is considered an abort. When the aborted thread resumes, it tries the atomic execution again from the beginning.

This approach has some very nice properties: no conflict resolution is needed, commits take no work, and it is easily integrated into an existing runtime. It also has obvious limitations. The model is explicitly designed only for the uniprocessor case, so atomic executions are conceptually equivalent to critical sections for which the thread scheduler holds one global lock. Furthermore, an atomic execution which exceeds its time slice will always be rolled back, even if it had no potential to interfere with other threads.

STM Monad

The STM monad is an extension of Haskell which offers monadic access to software transactional memory [HMJH05]. Haskell differs from every other programming language in the TM domain because it uses static typing and monads to identify and control side effects. In the case of STM, the monad offers transactional memory references, which can only be read or written from within the monad.

Haskell's approach offers more complex language-level abstractions for using atomicity. In addition to transactional reads and writes, the STM monad offers two more capabilities. The keyword `retry` forces the enclosing transaction to abort, keeping the log of the memory references touched during the execution. The transaction is retried only when some location in the log is changed. In this way, a transaction will not be re-run if its new execution would produce the same result as the old. The keyword `orElse` is a operator on two transactions; it runs one transaction, and if that transaction retries, immediately tries the other transaction rather than waiting. If that transaction also retries, the whole combined transaction waits as described above.

Transactional memory can only be accessed from within the STM monad; therefore there is no way to access non-transactional memory within a transaction, and vice versa. Since side effects are already restricted by the type system, this is not an unusual obstacle for a Haskell programmer.

Transactional Events

Concurrent ML has been extended with support for *transactional events*, using a new combinator `thenEvt` to form atomic sequences of synchronizing operations [EDKG08]. This allows transactional execution to be smoothly integrated with the rest of Concurrent ML's event combinators. Though Concurrent ML is based on message passing, the compositional nature of the event combinators give it more structure than is seen in other messaging systems, so it could be considered a form of orchestration. Thus, Ora is closely related to this line of research.

3.3.2 Limitations of Transactional Memory

Research in transactional memory has generally favored implementation and optimization over theory and semantics. This is possibly a result of its initial intended purpose, as a direct competitor to locks; subsequent systems were expected to demonstrate comparable efficiency, or be discarded as unfit competitors.

Despite being simple to use, transactions have not yet become simple to reason about. There is a lack of conceptual clarity about the semantics of transactions, and about the properties that they are intended to maintain. Formal language semantics and formal definitions of transactional properties are rare in the transactional memory literature.

For the sake of efficiency, many transactional memory systems forbid transactions from executing inside of other transactions. Almost all transactional memory systems disallow concurrent activity within a transaction, though in many cases this is a failure of the language itself to express concurrency.

Weak/Strong Atomicity

There is disagreement over the scope of the protection that a transaction should provide. Some approaches guarantee only *weak atomicity*: transactions are protected from each other, but nontransactional memory accesses can freely interfere with transactional memory accesses. Others provide *strong atomicity*: memory accesses not within a transaction must still respect the atomicity of transactional accesses [MBL06]. Weak atomicity has obvious problems, including the ability to totally violate atomicity by using nontransactional operations to "share" information between separate transactions. However, weak atomicity also admits more efficient implementations, since nontransactional memory accesses can simply ignore transactions entirely and thus incur no performance penalty.

Nesting Transactions

Since atomic blocks are syntactically no different from any other code block, another question arises: could atomic blocks be nested, and if so, what would that mean? Clearly this would give rise to a tree of transactions at runtime; how would atomicity be enforced, and rollback executed? Most transactional memory systems already disallow parallelism within a transaction, so nested subtransactions are never running in parallel. Thus, execution proceeds as an inorder traversal of a transaction tree. To support rollback in this context, it suffices to provide "savepoints": a stack of partial rollback points for the beginning of each subtransaction. This approach continues to be the default in mainstream systems to date.[4]

Various lines of research have explored the idea of nested transactions with parallelism. Some recently developed transactional memory systems, such as NePaLTM and XCilk, allow unbounded nesting with unbounded parallelism [VWAT^{+}09][AFS08]. Another recently introduced system, called Xfork, allows concurrent nested transactions by using combinators to connect subtransactions, providing explicit joins and mutual exclusion [RW09].

[4] Though this approach is easier and usually more efficient to implement, it hinders a consistent understanding of the semantics of transactions. Concurrency within transactions is the key to understanding both nested transactions and the use of message passing within transactions.

Chapter 4

A General Principle of Concurrency Control

This chapter describes a novel and general principle of concurrency control. This principle is more appropriate to control of orchestration and pervasive concurrency than previous approaches such as locking or transactions, since it is not conceptually rooted in the sequential execution of statements.

The classical abstraction for concurrent program behavior is the *trace*: a sequence of discrete program events. The properties of many existing concurrency control mechanisms are expressed in terms of traces. Here, we will instead organize program events using *event graphs*, where each node in the graph is an event, and directed edges between the nodes represent the causal ordering of events. The graph is a representation of a partial order, rather than a total order. This shift of perspective is critical, because it reveals an important intuition: sequentiality is not essential to transactions.

We will begin with a formal definition of the partial order, and its correspondence to a graph. Subsequently we will see how the execution of a program, and its interaction with shared resources, creates events in the graph and adds edges to the graph. Then, we will consider some simple example programs, using them to provide the intuition for a new principle of concurrency control: identifying an event subset called an *atom*, where every edge that enters or leaves the atom must be shared by every other event in the atom. We will consider a series of examples, to explore the implications of this principle, ending with an example that illuminates some shortfalls in the simple definition of atoms, followed by a revised definition of atoms that takes these issues into account.

This is an informal presentation, eliding many subtleties. We will see the full formal representation of these ideas in Chapter 8.

We will use Orc programs as examples throughout this chapter, though the methodology applies equally well to other concurrent programming languages.

4.1 Representation of Concurrent Events

An execution of a concurrent program is represented by two components:

1. A set of events, E.
2. A strict partial order, $\prec$, which is the *causal order* over the events of E.

Recall that a strict partial order has two key properties: it is *asymmetric*, meaning $a \prec b$ implies $b \not\prec a$, and it is *transitive*, meaning $a \prec b$ and $b \prec c$ imply $a \prec c$.

Just as many program actions are elided from a trace, since they are internal to the program's operation and do not affect the state of any shared resource, the set E contains only a salient subset of the program events; many "internal" or "silent" events will be elided.

If a program is nondeterministic, running the program multiple times may result in different partial orders, with a different set of events E or a different causal ordering $\prec$ of those events. This is unlike other models, such as event structures[Win89], which represent all possible executions of a program with a single, more elaborate construct.

For ease of presentation, we represent events and their partial order graphically using an *event graph*: a directed graph containing a node for each event in E, and an edge from event a to event b whenever $a \prec b$. Since $\prec$ is a strict partial order, it follows that event graphs are acyclic.

Each topological sort of the event graph results in a trace. Thus, the event graph could be thought of as a compact summary of a set of traces, factoring out all of the arbitrary ordering decisions in the traces.

Henceforth, we say that an event b *sees* an event a iff $a \prec b$.

4.1.1 Causality in Programs

Consider a very simple Orc program:

```
val r = Ref(0)
r := r? + 1
```

Recall from Chapter 2 that `?` reads from a mutable reference and `:=` writes to a mutable reference. Figure 4.1 shows the event graph for a full execution of this program.

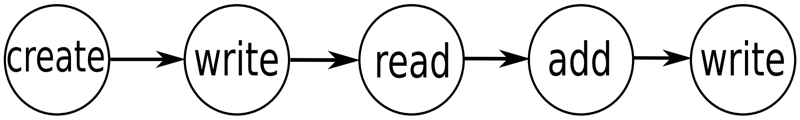


Figure 4.1: A simple event graph

Each node is a program event, and each edge represents the ordering of events required by the structure of the program. The first two events, **create** and **write**, represent the creation of the reference and its initialization to 0. Since the reference must be created before it can be written, the two events are ordered. All other events using the reference see both the **create** event and the **write** event.

Some internal events have been hidden because they are not relevant to the goal of representing and controlling concurrent interactions. Concurrent activity does not change the behavior of internal site call events, response events, or variable binding. Of course, these hidden events still provide a path of causes from one visible event to another; we use the transitivity of $\prec$ to replace the path with a single edge.

Consider another Orc program. This is essentially two copies of the previous program, running in parallel:

```
val r = Ref(0)
val s = Ref(0)
r := r? + 1 | s := s? + 1
```

Its associated event graph is given in Figure 4.2. No event from either subgraph sees an event from the other subgraph; each half of the program operates entirely in parallel, since each half is using a different reference.

Let's consider an example with some nondeterminism:

```
val b = true | false
Ift(b) >> Print("heads") | Iff(b) >> Print("tails")
```

There are two possible event graphs for this program, given in Figure 4.3 and Figure 4.4. In each case, one of the branches is truncated, since the call to `Iff` or `Ift` halts on an argument of `true` or `false`, respectively, so the corresponding **print** event never occurs.

4.1.2 Causality at Sites

Edges in the event graph are created by the program, and they can also be added by interactions with shared resources. Each of the sites that Orc provides to create shared resources has its own rules for the creation of edges. We will focus on the sites `Ref`, `Cell`, `Channel`, and `Semaphore`, which were introduced in Section 2.2.9.

For all resources, there is a **create** event that corresponds to the site call that created the resource. All operations on a resource see this event.

Each resource maintains an additional invariant: each event that sees the resource must be able to determine a unique state for that resource. Different observers may see different states for the same resource, but a single observer must not see more than one possible state. This property is called *consistency*.

Ref

The `Ref` site creates mutable references. There are two operations on these references: **read** and **write**. These operations obey two rules:

- There is a total order on all **write** events on a reference.

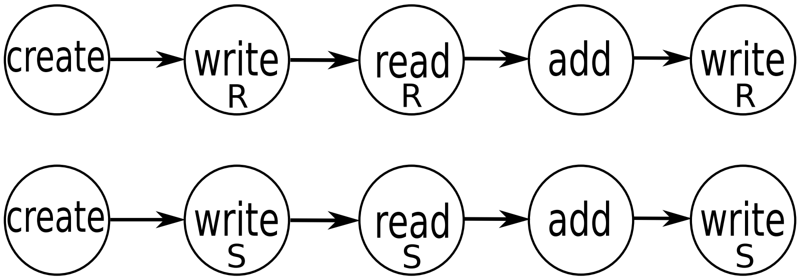


Figure 4.2: A simple disconnected event graph

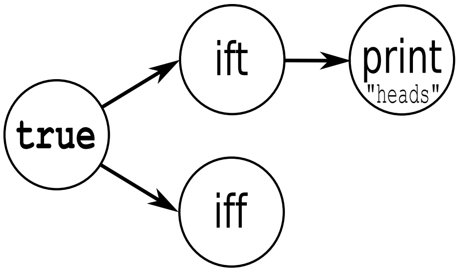


Figure 4.3: Nondeterministic binding, where b = `true`

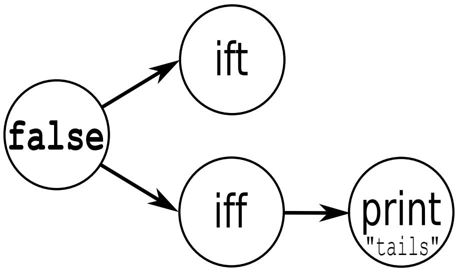


Figure 4.4: Nondeterministic binding, where b = `false`

- Every **read** event on a reference sees at least one **write** event on that reference.

When a reference is read, the **read** event sees some subset of the **write** events on that reference. Since **write** events are totally ordered, the subset has a unique maximum. The value read from the reference is the value that was written by that maximum **write** event.

`Cell`

The `Cell` site creates write-once mutable cells. Like references, there are two operations on cells: **read** and **write**. These operations obey two rules:

- There is at most one **write** event on a cell.
- Every **read** event on a cell sees exactly one **write** event on that cell.

When a cell is read, that **read** event sees the value written by the unique **write** event on that cell.

`Channel`

The `Channel` site creates FIFO asynchronous channels. There are two operations on channels: **put** and **get**. These operations obey three rules:

- There is a total order on all **put** events on a channel.
- There is a total order on all **get** events on a channel.
- Every **get** event on a channel sees more **put** events than **get** events.

Since **get** events and **put** events for a channel are each totally ordered, we can assign any given **get** or **put** event an index i. The ith **get** event on a channel receives the value that was sent by the ith **put** event on that channel. Since a **get** event sees more **put** events than **get** events, the ith **get** will always see at least i **put** events, so there is always an ith **put** available.

`Semaphore`

The `Semaphore` site creates semaphores. There are two operations on semaphores: **acquire** and **release**. These operations obey two rules:

- There is a total order on all **acquire** events on a semaphore.
- Every **acquire** event on a reference sees fewer **acquire** events than **release** events.

4.2 Controlling Concurrent Events

Let's consider a classic example: two parallel increments to the same reference.

```
val r = Ref(0)
r := r? + 1  |  r := r? + 1
```

What event graphs might result from the execution of this program? There are two nontrivial possibilities, shown in Figure 4.5 and Figure 4.6. Every other possible graph is equivalent to one of these two graphs. Recall that writes must be totally ordered, which restricts the space of possible graphs.

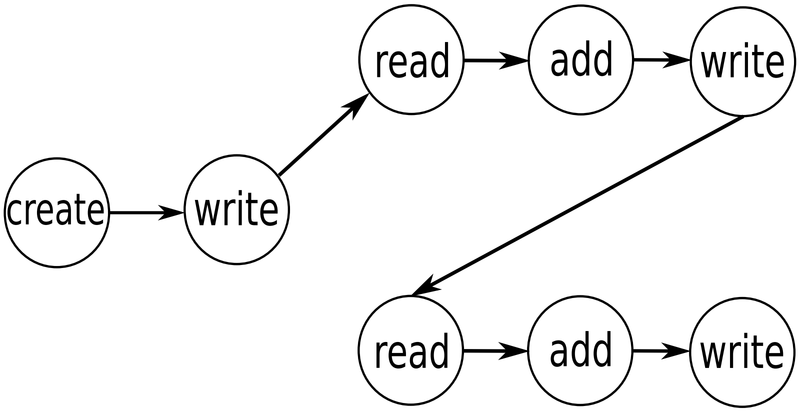


Figure 4.5: Two increments, sequenced

In Figure 4.5, the reference is read as 0, that value is incremented to 1, and then the value 1 is written. Subsequently the reference is read again as 1, that value is incremented to 2, and the value 2 is written. If the expression `r := r? + 1` is thought of as a single "increment" operation, then in this case, two such operations occur in sequence.

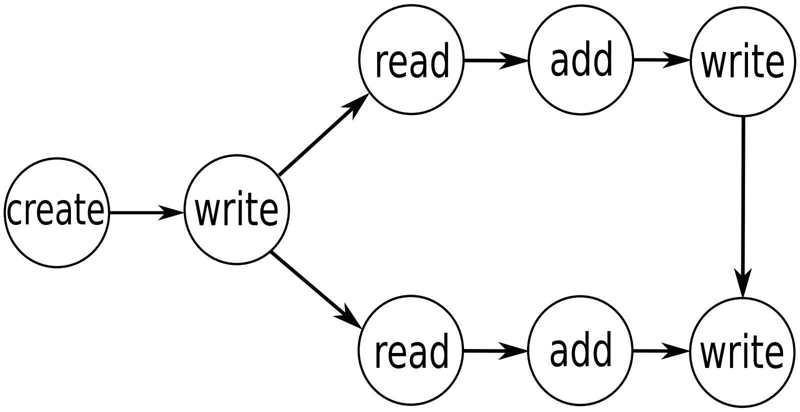


Figure 4.6: Two increments, lost update

In Figure 4.6, the reference is first read as 0, and that value is incremented to 1. In parallel, the reference is again read as 0, and that value is separately incremented to 1. Then two writes occur, in some order, each writing the value 1. If we again think of `r := r? + 1` as a single increment operation, then this behavior is confusing; two increments have been performed but the reference value has only increased by 1. This is a classic race condition, known as a "lost update".

We would like to express the idea that `r := r? + 1` is an indivisible operation. In the classic representation of program events using traces, we would simply say that there must exist an equivalent permutation of the trace where the read, add, and write events are contiguous. What requirement might we make of the event graph, to ensure that the events of `r := r? + 1` constitute an indivisible operation?

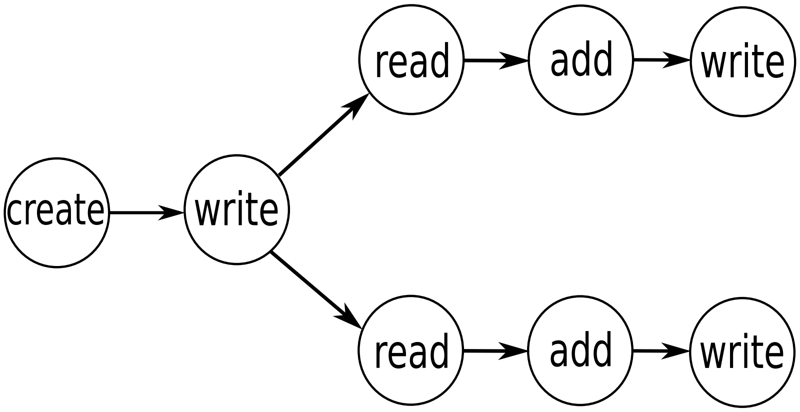


Figure 4.7: Two increments, divergent (*illegal*)

Note that the event graph shown in Figure 4.7 is *not* possible, since writes must be totally ordered.

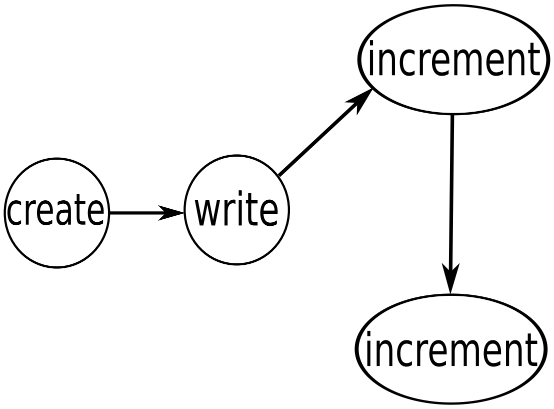


Figure 4.8: Figure 4.5, collapsed

Consider Figure 4.8, a collapsed version of Figure 4.5, where each subgraph corresponding to an increment operation has been replaced by a new **increment** event. This is the kind of graph we would like to reason about; the units we consider indivisible have been replaced with single events, so that we need not be concerned with their internal activity.

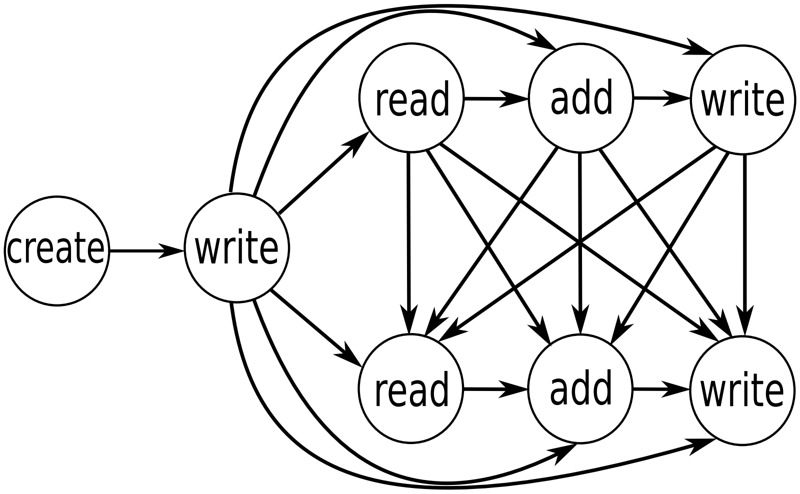


Figure 4.9: Figure 4.8, reexpanded

Now suppose we take the graph in Figure 4.8, and expand it again, replacing each **increment** event with the subgraph that it replaced. Each edge with its origin or destination at an **increment** event is copied for each event in the subgraph that replaces it. This proliferates many new edges. The expanded graph is shown in Figure 4.9.

If we take the graph in Figure 4.9, and remove edges implied by transitivity, then the result is exactly the graph in Figure 4.5. So, in this case, the collapsed and expanded graphs are equivalent representations of the program's behavior.

What happens if we repeat the same procedure with Figure 4.6? There are two possibilities, depending on whether we draw an edge from one **increment** event to the other in the collapsed graph. If we do draw the edge, then the collapsed graph is just Figure 4.8, and its expansion will just be Figure 4.5, which is different from the original graph. If we do not draw the edge, then the subsequent expansion will be equivalent to Figure 4.7, an invalid event graph. The condensed and expanded versions of Figure 4.6 are not equivalent representations. This makes sense, because the program behavior represented in Figure 4.6 is not what we want.

Whenever a subgraph has the property that each edge entering or leaving it is shared by every node in the subgraph, then condensing and expanding will produce equivalent representations. We can use this as the basis for defining the "indivisibility" of a set of events.

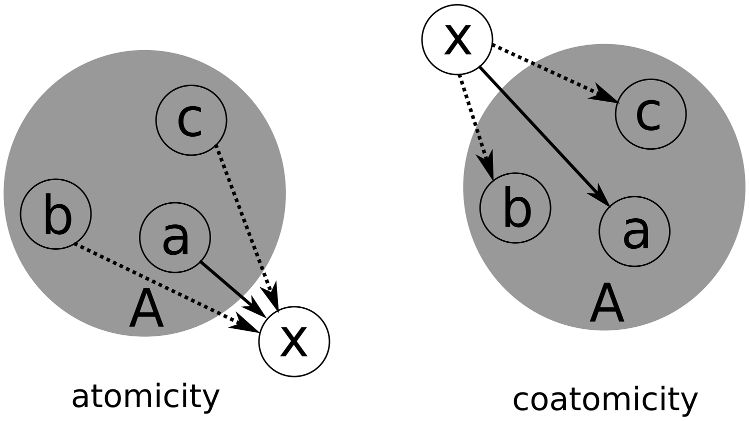


Figure 4.10: Atomicity and Coatomicity

4.2.1 Defining Atoms

Call any subset of events A an *atom* if the following two properties hold, for all events $a \in A$ and $x \notin A$:

Atomicity: If x sees a, then x sees every event in A.
That is, $a \prec x \implies \forall b : b \in A : b \prec x$

Coatomicity: If a sees x, then every event in A sees x.
That is, $x \prec a \implies \forall b : b \in A : x \prec b$

Figure 4.10 depicts these properties graphically.

The atomicity property ensures that an outside observer sees either all of the events in the atom, or none of them; no intermediate state is visible. The coatomicity property ensures that within the atom, every event has an equivalent view of the events outside the atom; it is as if the world outside of the atom has been frozen.

4.2.2 Examples

To understand these properties better, let's consider a few simple examples. In these example programs, we'll mark some expressions using the keyword **`atomic`**. The set of events in the execution of an **`atomic`** expression is marked by a gray box in the event graph. Each gray box must be an atom, as defined above; any event graph in which the events do not satisfy atomicity or coatomicity is considered invalid.

The implementation of **atomic**, and its implications for writing Orc programs, are discussed in subsequent chapters. For now, we will simply assume that it works as specified.

Write Twice

Let's consider a very simple example of atomicity.

```
val r = Ref(true)
r?  <<  atomic (r := false >> r := true)
```

This program creates a reference with the initial value `true`. An **atomic** expression sets the reference to `false`, and then back to `true`. In parallel, the reference is read. Since the writes are performed as an atom, the read should never see the intermediate state, so it should always publish `true`.

What event graphs might be produced by this program, and which of these are permitted by atomicity?

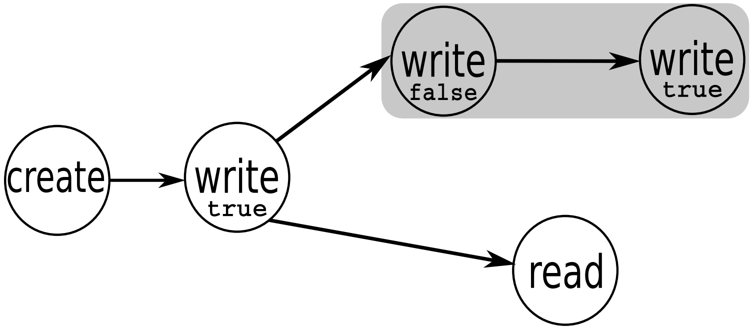


Figure 4.11: Both writes are unseen

In Figure 4.11, the read event sees only the initializing write; it does not see the other writes. Thus the read publishes `true`, the initial value of the reference. Atomicity holds trivially, since there are no edges leaving the atom.

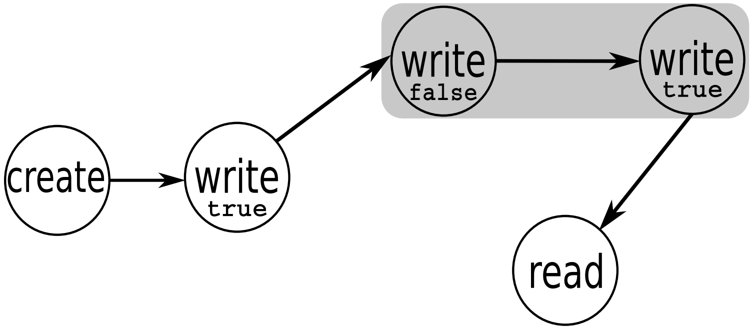


Figure 4.12: Both writes are seen

In Figure 4.12, the read sees both writes. It publishes `true`, because the maximum write it sees is `true`. Atomicity holds because the read sees both events in the atom: one directly, one by transitivity.

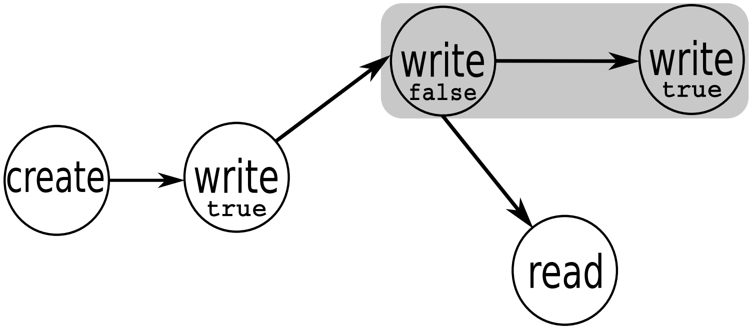


Figure 4.13: One write seen, one unseen

In Figure 4.13, the read event sees only the first write. It publishes `false`, since the write with `false` is the maximum write that it sees. However, in this case, atomicity is violated, since the read event sees one event in the atom (the `false` write), but does not see the other (the `true` write). Thus, this graph is not permitted by atomicity.

Note that in all cases, coatomicity holds trivially, since the **create** event and the initializing write are seen by both writes, and the read event is never seen by a write.

It follows that the program will always publish `true`.

Read Twice

Now let's consider an equally simple example of coatomicity.

```
val r = Ref(false)
atomic (r?, r?)  <<  r := true
```

This program creates a reference with the initial value `false`. Subsequently an atomic expression reads the reference twice, creating a tuple of the values read. In parallel, a write operation sets the reference to `true`. Since both reads occur within an atom, each read should see the same external state, so the tuple must always have equal elements: `(false, false)` or `(true, true)`.

What event graphs might be produced by this program, and which of these are permitted by coatomicity?

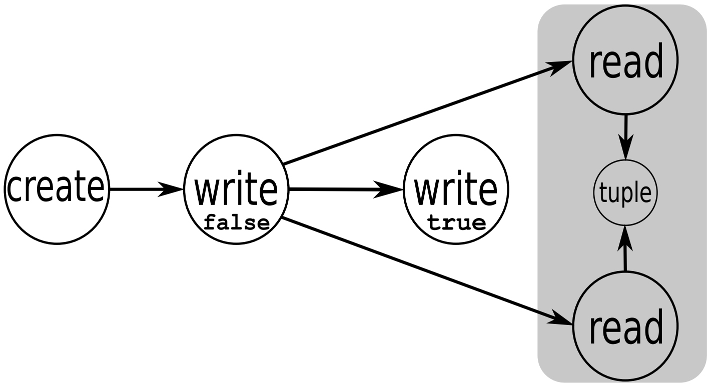


Figure 4.14: Both reads miss the write

In Figure 4.14, neither read event sees the write of `true`. Thus each read publishes `false`, the value set by the initial write, which is the only write each read sees. The resulting tuple is
`(false, false)`. Coatomicity holds, since both reads see the initial write, the **tuple** event sees the initial write transitively, and no other edges enter the atom.

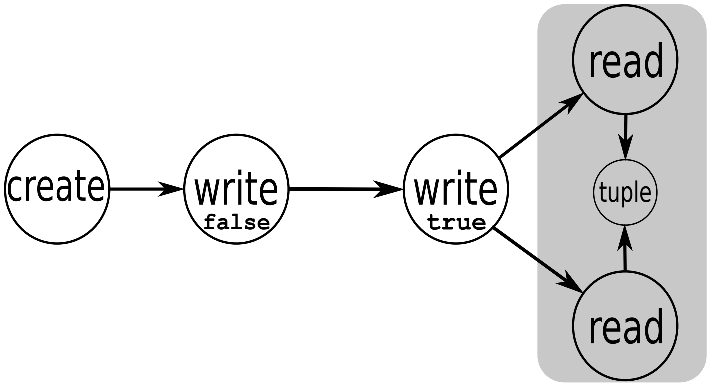


Figure 4.15: Both reads see the write

In Figure 4.15, both read events see the write event. Thus each read publishes `true`, the maximum visible write; the result is `(true, true)`. Coatomicity holds, since each write is seen by every event in the atom.

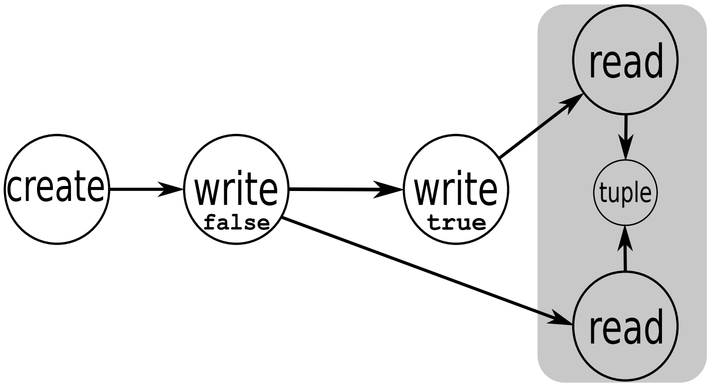


Figure 4.16: Only one read sees the write

In Figure 4.16, only one read event sees the write event. Consequently, the reads publish different values: `true` and `false`. However, coatomicity fails, since the write is seen by one event in the atom and not also seen by the other event.

Note that in all cases, atomicity holds trivially, since there are never any edges leaving the atom. It follows that the program will only ever publish `(false, false)` or `(true, true)`.

Double Increment

Atoms are not required to be disjoint; they may be nested within other atoms, to arbitrary depth. Here we consider an example where two atoms are nested inside of a larger atom.

```
val r = Ref(0)
r? << atomic ( atomic (r := r? + 1) & atomic (r := r? + 1) )
```

This program resembles the two-increment program we considered earlier. It creates a reference with an initial value of zero, and then performs two increment operations on that reference. This time, there is also a read operation on the reference, in parallel with the writes.

Unlike the earlier example, we have explicitly marked each increment operation as an atom, to avoid the 'lost update' problem. In addition, the two increments together are also marked as an atom. Thus, any outside event will either see both increments, or neither. In particular, the read operation `r?` should publish only `0` or `2`.

What event graphs might be produced by this program, and which of these are permitted by atomicity and coatomicity?

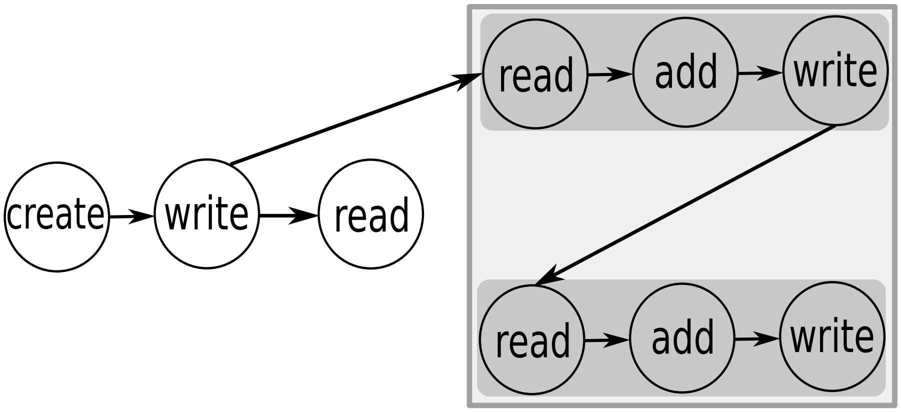


Figure 4.17: Two increments, unseen

In Figure 4.17, the two increments in the atom happen in sequence; the terminal **write** of one increment is seen by the initial **read** of the other increment. Thus, atomicity and coatomicity are satisfied by transitivity for each internal atom, and there is no lost update. Every event in the enclosing atom sees the initializing **write**, and the external **read** sees no events from the atom, so both atomicity and coatomicity are satisfied for the enclosing atom as well. The program publishes `0`.

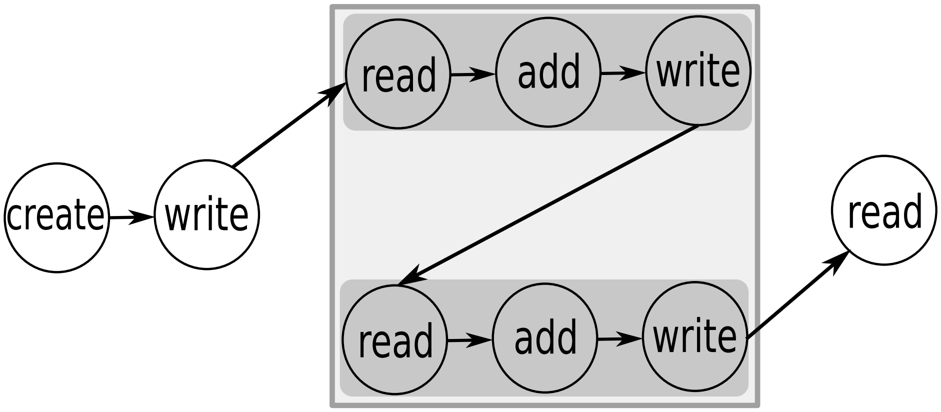


Figure 4.18: Two increments, seen

In Figure 4.18, the two increments in the atom happen in sequence, satisfying atomicity and coatomicity as before. Every event in the enclosing atom sees the initializing **write**. The external **read** sees the **write** from the second increment, and so it sees every event in the enclosing atom, by transitivity. Thus, both atomicity and coatomicity are satisfied for the enclosing atom as well. The program publishes `2`.

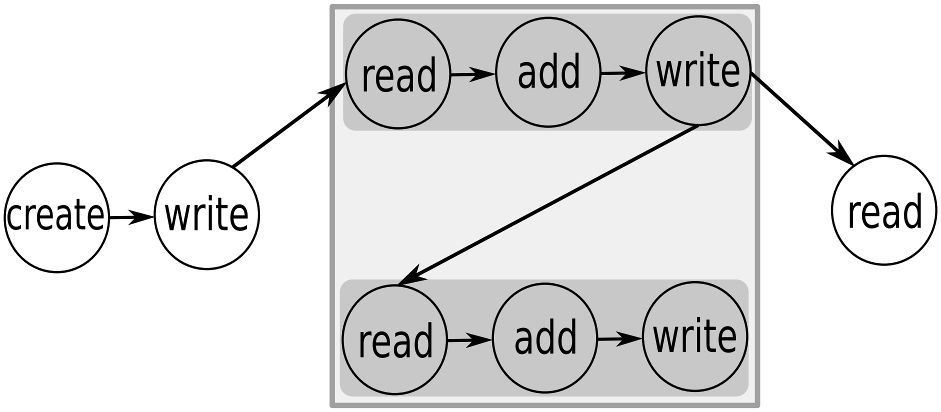


Figure 4.19: Two increments, partial visibility (invalid)

In Figure 4.19, the two increments in the atom happen in sequence, and atomicity and coatomicity are satisfied as in the previous graphs. However, this time the external **read** sees the **write** from the first increment, and not the second; as a result, it does not see the events of the second increment, and so atomicity is violated for the enclosing atom. In this case the program would publish `1`, but this execution is invalid and so it would never occur.

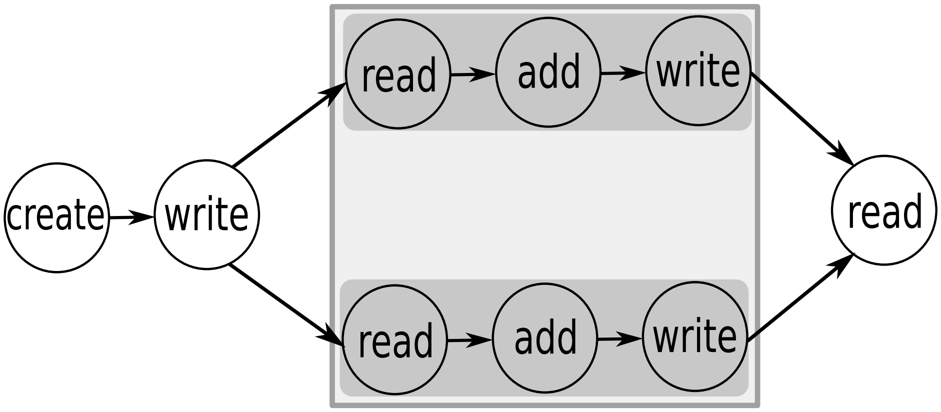


Figure 4.20: Divergent writes (invalid)

Figure 4.20 shows another invalid execution. This time, the invariants on **write** events are not satisfied. Writes must be totally ordered, but the writes in each increment are not ordered with respect to each other. As a result, the read event could see multiple possible states for the reference, violating the consistency property defined in Section 4.1.2. Thus, this execution can never occur.

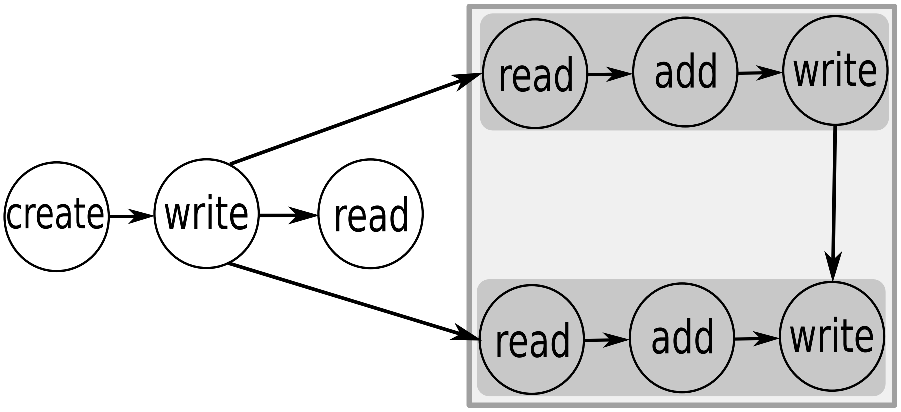


Figure 4.21: Internal lost update (invalid)

Figure 4.21 shows one more invalid execution. This is the ‘lost update’ problem described earlier; this configuration violates coatomicity of one of the internal atoms, since the **write** should be seen by every event in that atom, but the **read** does not see it.

Reads and Writes

Let’s consider a more complex example, which will illustrate some weaknesses in the simple definition of atoms that we have used thus far.

```
val r = Ref(false)
val s = Ref(false)
atomic (r?, s?)  <<  atomic (r := true  &  s := true)
```

This program creates two references, both set to an initial value of `false`. It then performs two operations in parallel: set both references to `true`, and read the references as a tuple. Each of these operations is an atom. Thus, the reads should each see the same state: both references containing `false`, or both containing `true`.

Note that if either of the operations were not an atom, the result could be `(true, false)` or `(false, true)`. If the reads were not an atom, one read could see a reference state before the writes, and one after, leading to an inconsistent tuple. If the writes were not an atom, the reads could see a state with one reference written, but not the other, also leading to an inconsistent tuple.

What event graphs might be produced by this program, and which of these are permitted by atomicity and coatomicity?

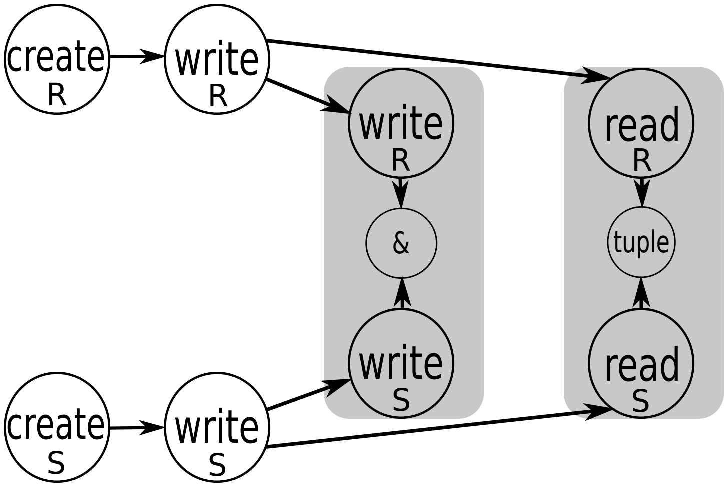


Figure 4.22: Both writes unseen

In Figure 4.22, neither read event in `(r?, s?)` sees either write from the other atom. Thus, each read sees the initial value from its respective reference, forming the tuple `(false, false)`.

The program's behavior seems to agree with our expectations. However, coatomicity is technically not satisfied by this graph; there are edges implied by coatomicity that are absent here. Figure 4.23 shows the graph with the edges needed for coatomicity added as dotted lines.

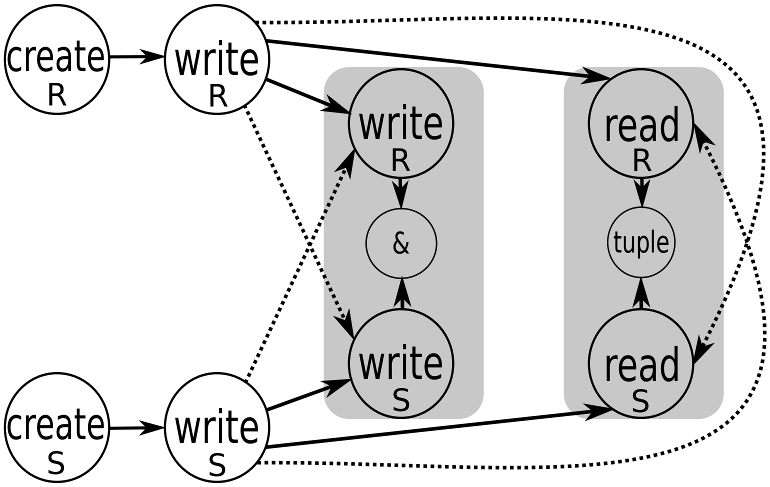


Figure 4.23: Both writes unseen, missing edges added

As far as the program's behavior is concerned, these edges are irrelevant. Each of the edges connects an event at one reference to an event at another, but the behavior of an event on one reference can't be affected by events at other references. The behavior of a read at reference R will be the same regardless of which reads or writes at reference S it sees or does not see.

So, the definitions of atomicity and coatomicity will need a small revision: if an edge could never affect the behavior of the program, its absence does not violate atomicity or coatomicity.

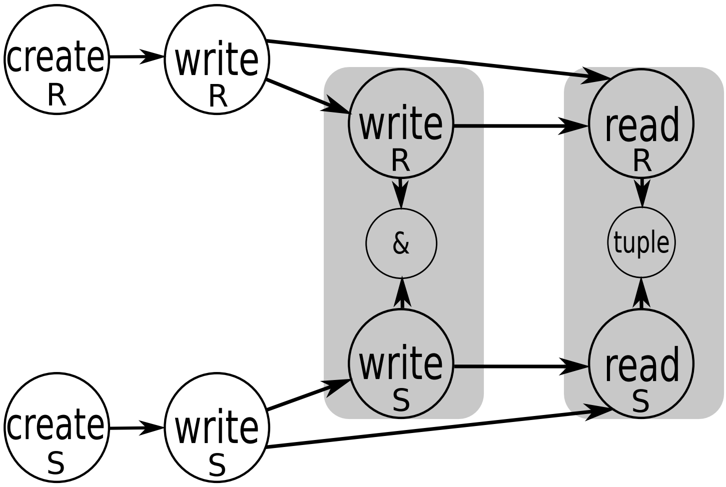


Figure 4.24: Both writes seen

In Figure 4.24, both read events in `(r?, s?)` see their corresponding writes from the other atom. Thus, each read sees the newly written value `true` from its respective reference, forming the tuple `(true, true)`.

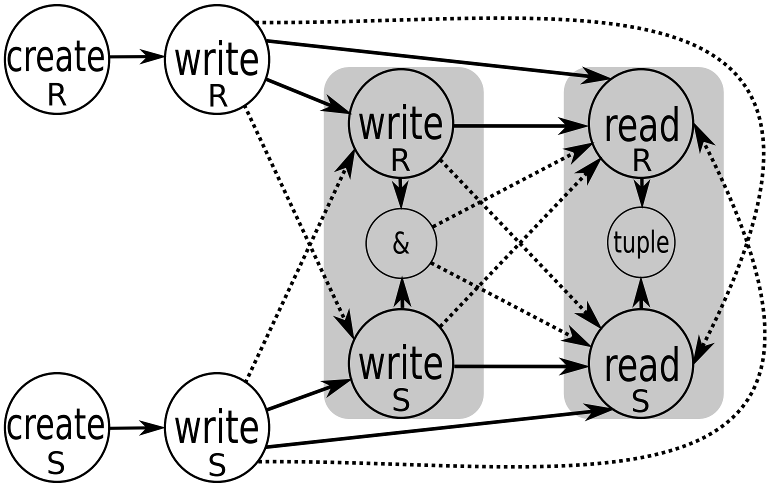


Figure 4.25: Both writes seen, missing edges added

Just as before, there are edges missing. Figure 4.25 shows the graph with the edges required by atomicity and coatomicity added as dotted lines. The edges from the previous case are present, as well as a new set of edges connecting the two atoms. In addition to edges that connect events on unrelated references, there are also edges that connect the & operation, a pure site call, to the read events. These edges are all irrelevant. An & operation has no effect on other events of any kind, just like + or < or any other mathematical operation. And as before, edges between events on different references are irrelevant.

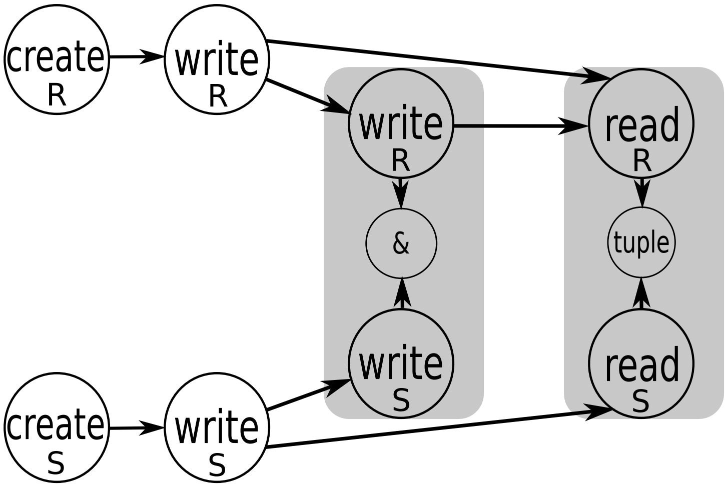


Figure 4.26: Only one write seen

In Figure 4.26, only one of the read events in `(r?, s?)` sees its corresponding write from the other atom. Thus, one read sees the newly written value `true`, but the other sees the initial value `false`, forming the tuple `(true, false)`.

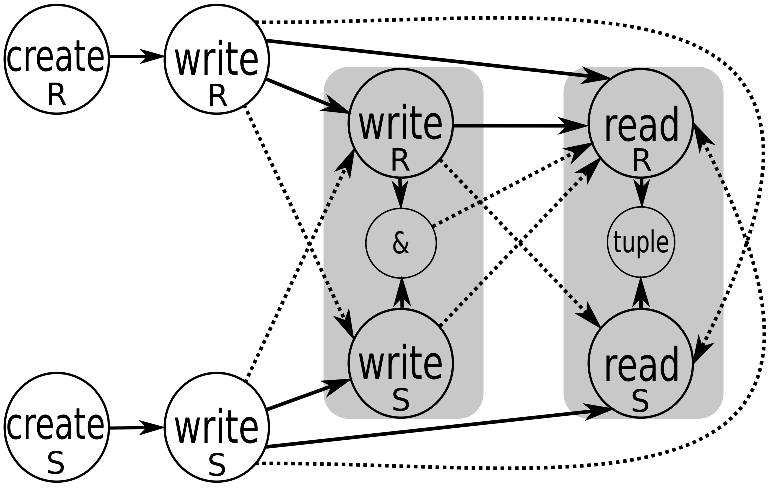


Figure 4.27: Only one write seen, missing edges added

Adding the missing edges needed for atomicity and coatomicity gives us Figure 4.27. As before, each of these edges is irrelevant to the program's behavior. But if we ignore the absence of those edges, we are left with a program that appears to obey atomicity and coatomicity, but produces behavior that seems to be incorrect. What has gone wrong?

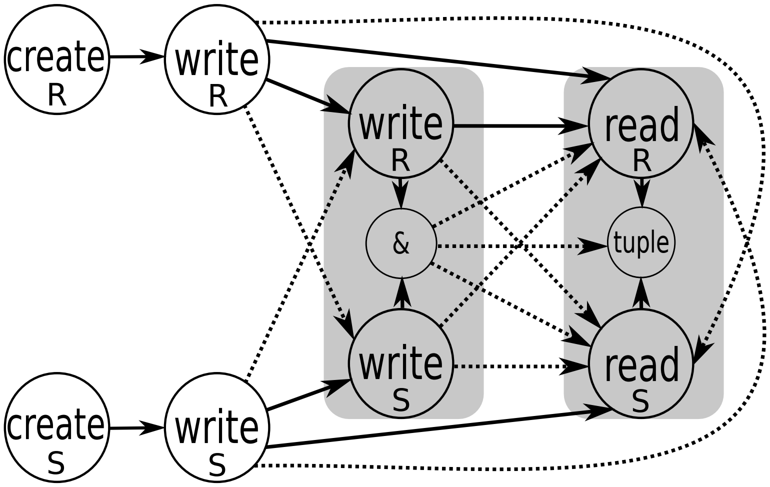


Figure 4.28: Only one write seen, least fixed point

Consider what would happen if we took Figure 4.27, and continued adding new edges implied by atomicity and coatomicity. In other words, what if we took the least fixed point of the atomicity and coatomicity rules? The result is Figure 4.28.

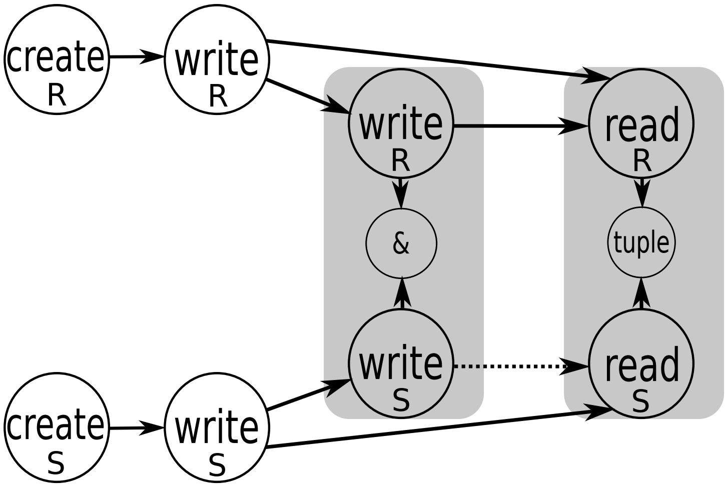


Figure 4.29: Only one write seen, missing edge identified

Now, as before, we discard each of the irrelevant edges, resulting in Figure 4.29. One edge remains: an implied edge from the unseen write in one atom to the corresponding read in the other atom. This edge does affect the behavior of the program, but it was not present in the original event graph. The absence of that implied edge should indicate that atomicity or coatomicity has been violated, but in order to correctly account for this situation, we need to revise the definition of an atom.

4.2.3 Redefining Atoms

In order to account for the issues raised by the example in the previous section, we will need to revise the definition of an atom.

We will approach the problem from a different perspective: suppose we are given a set of subsets $\mathcal{A}$ of the entire event set E, and told that each subset $A \in \mathcal{A}$ is an atom. How would we validate this choice of atoms?

We will say that an event x is *relevant* to an event y if $x \prec y$ is possible in *some* execution, even if it is not true in the current execution.

We define a new strict partial order $\textcircled{$\prec$}$, the *virtual order*, which represents the requirements of atomicity and coatomicity. The dotted edges in the preceding event graphs are a graphical representation of $\textcircled{$\prec$}$. The $\textcircled{$\prec$}$ relation has three rules:

1. Whenever $x \prec y$, $x \textcircled{$\prec$} y$ also. That is, we treat solid edges as if they were also dotted edges.

2. Whenever we have events $a \in A$, $b \in A$, and $x \notin A$, and we know $a \textcircled{$\prec$} x$, then $b \textcircled{$\prec$} x$ also. This expresses atomicity.

3. Whenever we have events $a \in A$, $b \in A$, and $x \notin A$, and we know $x \textcircled{$\prec$} a$, then $x \textcircled{$\prec$} b$ also. This expresses coatomicity.

We apply these rules repeatedly, until $\textcircled{$\prec$}$ cannot be expanded any further. Then the selection of atoms is valid if whenever x is relevant to y, and $x \textcircled{$\prec$} y$ holds, then $x \prec y$ also holds.

We will see a more formal presentation of this concept in Section 8.1.

4.3 Related Work

While the presentation of atomicity and coatomicity as symmetric properties over partial orders is a novel approach, the pursuit of a formal classification or definition for transactional operations has been ongoing for decades. The concept shown here has many antecedents in the transactional literature.

In the same paper that defined the concepts of weak atomicity and strong atomicity [MBL06], Blundell et al. also mention the concepts of "noninterference" and "containment" to elucidate the interaction of certain transactions. These concepts could be seen as analogous in function to coatomicity and atomicity respectively. However, the paper does not explain these concepts further nor define them formally, and does not consider their applicability to nested or internally concurrent transactions.

Farzan and Madhusudan propose a notion of *causal atomicity*, which moves the focus to partial ordering of events, and defines atomicity in terms of that partial ordering [FM06]. The subsequent analysis is presented in terms of colored Petri nets. However, the focus is on program analysis and the detection of atomicity, not on language integration. Again, nested and internally concurrent transactions are not considered.

The general principle described in this chapter also closely parallels the fundamental ACID properties of database transactions: Atomicity, Consistency, Isolation, and Durability.

The database property of Atomicity maps almost directly to the atomicity property described in this chapter: an all-or-none view of events. The property of Consistency also corresponds closely with the notion of consistency described here, in the sense that the appropriate invariants of shared resources must be preserved in order to ensure that no resource is left in an invalid or indeterminate state. The property of Durability is not essential in this context; many of the shared resources that could be used by a concurrent program are not associated with durable storage and do not need to be.

The most interesting point of comparison is between the Isolation property of database transactions and the coatomicity property described in this chapter. Definitions and implementations of isolation vary widely. As a reference point, consider the transaction isolation levels defined in the SQL standard: Read Uncommitted, Read Committed, Repeatable Read, and Serializable [SQL92].

Read Uncommitted allows operations that would violate atomicity, so it is clearly too weak to be comparable. Read Committed allows operations that would violate coatomicity, since it allows a transactional read to see a write not seen by another read within the same transaction (a "non-repeatable read"). Repeatable Read also allows operations that would violate coatomicity, though the violation is more subtle: since coatomicity would not distinguish between events that modify records and events that add records, the "phantom read" operations permitted by a Repeatable Read isolation level are still not coatomic. An isolation level of Serializable seems most comparable to coatomicity. If transactions can be totally ordered, then both atomicity and coatomicity are assured: each event of an earlier transaction is certain to precede every event of a later transaction. In fact, Serializable isolation is a *stronger* constraint than coatomicity, since coatomicity does not require equivalence to some serial order on the atoms or events it constrains. We will discuss nonserializable executions further in Section 9.4.

Chapter 5

Ora

This chapter focuses on the integration of the concurrency control principles of atomicity and coatomicity into the Orc programming language. Orc programs are pervasively and often implicitly concurrent, and while this abundance of concurrency is usually beneficial, it compounds the classic problems of concurrency control.

This chapter focuses on Ora, an extended version of the Orc language with a new combinator called **atomic** that defines atoms, as explained in the previous chapter.[1] We will first discuss the behavior of the **atomic** combinator, and how it integrates with the rest of the Orc language. Then, we will see how the availability of **atomic** provides a powerful new capability called *atomic choice*, a generalization of the guarded choice operator in the π calculus [Mil99]. Subsequently we will consider a series of Ora programs, examining how **atomic** and atomic choice provide concurrency control solutions for various common problems that arise when writing concurrent programs, particularly when writing Orc programs.

[1] Ora is a contraction of “Orc with **atomic**”. It is also a play on *ōra*, which in Latin means “boundary” or “region”, referring to its fundamental concurrency control principle: to define boundaries of atomic activity.

5.1 The atomic combinator

The Ora language extends Orc with a new combinator, **atomic** *f g*.

When **atomic** *f g* is executed, it executes the expression *f* as an atom. That is, every event that occurs in *f* must obey the constraints of atomicity and coatomicity, as described in Chapter 4. The execution of *f* also operates under a few additional constraints, which ensure that the program's internal events satisfy atomicity and coatomicity:

1. *f* begins to execute only when it has no free variables.

2. No event outside of *f* may see any event in *f* until *f* halts.

3. If *f* would publish a value, that value is instead *frozen*. When *f* halts, all frozen values are published simultaneously. If *f* is killed for any reason, the frozen values are discarded, and never published.

The execution of *f* may reach a state where consistency could be violated if some event that occured in *f* is ever seen outside of *f*. If this happens, then *f* is killed, and *g* runs; this is called an *abort*. Note that *g* does not execute as an atom, nor does it have any of the other constraints that are applied to *f*.

5.1.1 Unary atomic

The expression *g* is used as a fallback in case *f* does not succeed. However, it is often desirable to just retry *f* when it fails, rather than executing a new expression, and in that case a second argument is not needed. A unary form of **atomic** is simple to express in terms of the binary form. Define **atomic** *f* as follows:

```
                def p() = atomic f p()
atomic f   ≡    p()
```

5.1.2 The Abort Site

It is sometimes useful to explicitly force an abort, even if an **atomic** expression has not violated atomicity, coatomicity, or consistency. Ora introduces a new site `Abort` for this purpose. A call to `Abort` takes no arguments. If during the execution of **atomic** *f g*, `Abort` is called in *f*, then *f* is killed and *g* executes, just as if *f* had aborted normally. If `Abort` is called without an enclosing **atomic** expression, the call simply halts.

5.2 Atomic Choice

The π-calculus has a choice operator $+$, which permits a nondeterministic choice between two processes. In its simplest form, this choice is unrestricted: the π-calculus expression $P + Q$ becomes either P or Q in one step. This is easy to replicate in Orc:

$$P+Q \quad \equiv \quad \textbf{if } \texttt{(true | false)} \textbf{ then } P \textbf{ else } Q$$

However, this is rarely useful in practice, because the choice is entirely arbitrary; it is not driven by events in the environment or by the availability of messages or resources. Depending on the implementation, the choice may not even be fair.

So instead of unrestricted choice, the π-calculus often uses *guarded choice.* In a guarded choice $P + Q$, the processes P and Q must each begin with some synchronizing action, such as a channel input. The choice is then guarded by this action; a process may only be chosen if its initial action can be performed. For example, here is a process that waits for the first input seen from channel a or b, and then sends it to channel c:

$$a?(x).c!(x) + b?(x).c!(x)$$

We would like to express this as an Ora program. Here is a naive initial attempt:

```
val x = a.get() | b.get()
c.put(x)
```

The calls `a.get()` and `b.get()` occur in parallel. Suppose an input value v becomes available on channel a. Then `a.get()` returns v, which is then published and bound to `x`, terminating `b.get()`. Then `c.put(x)` sends v on `c`.

However, this program is *not* equivalent to the π-calculus version. When a site call is terminated by the pruning combinator, the call still continues at the site, uninterrupted; its eventual return value is simply ignored. Thus, `b.get()` will still take a value from the channel b if one becomes available, and then discard it.

The difficulty seems to arise because pruning may occur between a call and its return. What if we used **atomic** to make the call and return happen in one indivisible step?

```
val x = atomic a.get() | atomic b.get()
c.put(x)
```

Unfortunately, this also does not solve the problem. Even though we have an 'all or none' view of each call, such that it is not possible to see a channel call without its associated return, we can still 'waste' a value if *both* atomic executions successfully complete before the pruning occurs. Then the pruning combinator picks only one of the resulting publications, wasting the other. Ora has no scheduling constraints to prevent this scenario.

Guard

To solve this problem, we augment Ora with a new site called Guard, which creates *guards*. A guard G is a site that returns a **signal** to its first caller, and then halts on all subsequent calls; it returns at most once. It explicitly represents the intent to choose only one branch.

Using a guard, we represent guarded choice in the following way:

```
Guard() >G>
 ( atomic (G() >> a.get())
 | atomic (G() >> b.get()) )
```

In this example, expressions G() >> a.get() and G() >> b.get() each execute atomically. The call G() returns a signal in *both* executions. This seems to contradict the definition of a guard, but remember that due to atomicity, each return from G is not seen outside of its respective atomic execution, so no observer could see that G has returned twice. In fact, to any observer outside of these two atoms, G has never even been called. Note that if we did not use **atomic** here, the second call to G() would never return, and this would be equivalent to unguarded choice.

Both expressions cannot successfully *complete*, since then the effects of both expressions will be seen by the environment, and then an observer could see that G has returned twice, violating consistency. Therefore, only one of the two branches is allowed to complete; the other one never completes, and thus from the outside it appears that it never executed at all, again due to atomicity. Note that even when one of the two executions completes, and the guard call is seen by the environment, it is still not seen by the other expression, due to coatomicity. Therefore it is still true that no observer sees that `G` has returned more than once.

An expression will only complete if the `get` call succeeds, so the result is "chosen" based on the success of that action. Thus, at most one of `a.get()` or `b.get()` is chosen, depending on which channel has available input, and the unchosen expression appears never to run at all. This is semantically equivalent to the π-calculus expression shown earlier.

In fact, this *atomic choice* generalizes guarded choice, since we can choose based on a whole expression, rather than just a single action. As we will see in subsequent examples, this capability is quite powerful.

A perceptive reader might wonder why `Guard` is even necessary. Creating a `Cell` that both branches attempt to write would seem to have equivalent behavior. However, an implementation of `Cell` might see the competing writes as a possible violation of consistency, and preemptively abort one of the branches, before either one had halted. An arbitrary abort defeats the purpose of atomic choice.

The semantics of Ora, as described in detail in Chapter 7, do not allow an abort until a violation of consistency is inevitable. However, an alternative implementation of the Ora language might mediate potential conflicts differently. So, in order to ensure that `++` is implemented faithfully by its encoding, the `Guard` site is used to signify a need for the most conservative abort strategy possible.

The ++ Operator

Given Ora expressions A and B, we will subsequently use the operator `++` as syntactic sugar for atomic choice:

```
                    Guard() >G>
A ++ B      ≡        ( atomic ( G() >> A )
                     | atomic ( G() >> B ))
```

The `++` operator is a fully associative and commutative n-ary operator, as is evident from the definition above, since the parallel combinator is associative and commutative, and we can simply add more parallel clauses.

In particular, the following π-calculus process and Ora expression are equivalent:

$a?(x).c!(x) + b?(x).c!(x)$ ≡ `(a.get() ++ b.get()) >x> c.put(x)`

The **atomic** combinator enables encodings for many other concurrent primitives. For example, the multi-way join patterns of the join calculus could be encoded by associating a channel with each process abstraction; a call sends its arguments as a tuple on the corresponding channel, while a definition waits on multiple channels simultaneously within an **atomic** body, waiting for the needed message to arrive from each channel.[2]

These encodings for guarded choice and join patterns are similar to those presented in AtCCS, an extension of CCS with transactions [ABZ07], developed independently from Ora. However, the guarded choice encoding in AtCCS is less general than the atomic choice shown here, since Ora's ++ operator can make a choice based on the execution of an entire expression.

[2] Channels in Ora are ordered, so under this encoding, process abstractions would receive inputs in the order that they were sent. The join calculus does not require this kind of ordering; fortunately, it has little effect on the behavior of the encoding. An unordered channel primitive would be more appropriate, but the implementation of Ora described in this dissertation supports only ordered channels.

5.3 Writing Programs in Ora

This section presents a series of example programs, demonstrating how to solve common concurrency control problems using the **atomic** combinator and the ++ (atomic choice) operator. As with the other combinators of Orc, there are various programming tactics and idioms that are useful when working with **atomic**.

5.3.1 Account Transfer

This example is a classic problem from the transactional memory literature: a transfer between two bank accounts.

We model each account as a mutable reference. Here is a function `transfer`, which subtracts some amount from one account and adds it to another:

```
def transfer(x, y, amount) =
  atomic (
       x := x? - amount
    &  y := y? + amount
  )
```

Due to atomicity, we know that any observer outside the call to `transfer` will see either both account changes, or neither. As a result, the sum of the accounts is invariant.

However, this simple example needs some refinement. What happens if the source account has insufficient funds for the transfer? We need to check if `x` has at least a value of `amount`. As a sanity check, we will also make sure that `amount` is greater than 0.

```
def transfer(x, y, amount) =
  Ift(amount :> 0) >>
  atomic (
    Ift(x? >= amount) >>
    (
       x := x? - amount
    &  y := y? + amount
    )
  )
```

Note that the check on amount is outside of **atomic**; it is a pure operation, so it has no interactions with other events. However, the check on x is within **atomic**, since we must be sure that the value of x does not change between the two reads, to avoid setting x to a negative value.

Also note that the check on x uses Ift, which will simply halt if the amount is insufficient. This does not cause an abort; the **atomic** expression simply completes at that moment, halting without modifying either account. We interpret a halted call of transfer as an indication that the transfer did not occur. If the checks succeed and the transfer does occur, the call will instead publish **signal**.

This function needs one more refinement. What happens if the source and destination accounts are identical? Then the two account modifications could be interleaved, and we might end up with a "lost update", like the two-increments problem from Section 4.2. We could fix this by comparing the identity of the accounts, and publish **signal** without making a transfer if they are equal. Or, we could simply enclose each account modification in its own **atomic**, preventing a lost update. Here is the final version of transfer:

```
def transfer(x, y, amount) =
  atomic (
    Ift(x? >= amount) >>
    (
       atomic (x := x? - amount)
    &  atomic (y := y? + amount)
    )
  )
```

5.3.2 Permutation

This example focuses on a common problem in concurrent and distributed programming: obtaining a consistent view of a resource that changes over time. Here, we will see how **atomic** enables us to perform a computation on a list that is continuously being shuffled.

We first define a few helper functions. These are some typical list functions:

```
def nth(0, x:xs) = x
def nth(n, x:xs) = nth(n-1, xs)

def length([]) = 0
def length(x:xs) = 1 + length(xs)
```

`nth(n,xs)` publishes the `n`th element of list `xs`, starting at 0; if there is no `n`th element, the call halts. `length` publishes the number of elements in the list.

The following helper functions are specific to this example:

```
def atRandomIntervals(t) =
  Rwait(Random(t+1)) >> ( signal | atRandomIntervals(t) )

def randomItem(xs) =
  nth(Random(length(xs)), xs)

def total([]) = 0
def total(x:xs) = x? + total(xs)

def swap(x,y) =
  atomic ((x?, y?) >(vx, vy)> x := vy & y := vx)
```

`atRandomIntervals(t)` publishes **`signal`** repeatedly, spaced at random intervals of no less than 1 millisecond and no more than `t` milliseconds.

`randomItem(xs)` publishes a random element of the list `xs`. All elements are equally probable. If the list is empty, the call halts.

`total(xs)` takes a list `xs` containing mutable references. It reads each reference, and publishes the sum of their values.

`swap(x,y)` takes two mutable references `x` and `y`, and exchanges their values. Since it is enclosed by **`atomic`**, other events in the program will see the references in either their unswapped state or their swapped state, but never in an intermediate state.

```
val numbers =
  [Ref(1), Ref(2), Ref(3), Ref(4), Ref(5)]

atRandomIntervals(100) >>
  swap(randomItem(numbers), randomItem(numbers)) >> stop
|
atRandomIntervals(100) >>
  atomic ( total(numbers) )
```

This program assembles a list of references, each of which contains a number from 1 to 5. It then performs two activities in parallel, repeatedly, each at random intervals: swap the values stored at two random references from the list, and compute the total of the values in the list, publishing that value.

The correct behavior of the program is to repeatedly publish `15`, the total of the values stored in the references, since a swap never changes the total value of the list. Even though the `swap` function is atomic, that alone does not ensure the correctness of the program. If `total` were not called within **atomic**, then a swap operation during the execution of `total` might swap an unexamined element with an examined one, causing some number to be totaled twice, and another number to be missed. However, the use of **atomic** guarantees the correctness of `total`: due to coatomicity, *every* read operation in `total` must see the same set of writes. So, if the effect of a swap is seen by a read on one of the swapped references, it must also be seen by the read on the other swapped reference.

5.3.3 Dining Philosophers

This example presents the classic Dining Philosophers problem [Dij65], in Ora. The dining philosophers program uses locking; while transactions are usually considered to be an alternative to locks, in Ora we can use locks in the context of **atomic** just like any other resource. In particular, **atomic** provides a means to acquire a set of locks simultaneously, without requiring a global lock order or some equivalent tactic to prevent deadlock.

Here is the dining philosophers program in Ora:

```
def think() = Print("Thinking...") >> Rwait(Random(1000))
def eat() = Print("Eating...") >> Rwait(Random(1000))

def philosopher(l,r) =
  think() >>
  atomic (l.acquire() & r.acquire()) >>
  eat() >>
  atomic (l.release() & r.release()) >>
  philosopher(l,r)

val a = Semaphore(1)
val b = Semaphore(1)
val c = Semaphore(1)
val d = Semaphore(1)
val e = Semaphore(1)

  philosopher(a,b)
| philosopher(b,c)
| philosopher(c,d)
| philosopher(d,e)
| philosopher(e,a)
```

The program defines five instances of Semaphore, each of which acts as a fork. Each fork is shared by two calls to philosopher, a recursive function that simulates a philosopher process. The functions think and eat are simulations of the thinking and eating states of the philosopher. When a philosopher process wants to eat, it acquires its two forks, and when it is done it releases the forks.

To prevent the classic deadlock case, the **atomic** combinator protects the acquisition and release of the forks. No observer can see a state where a philosopher is holding only one fork. This prevents the formation of a cycle of waiting philosophers that would result in deadlock.

However, there is still a potential for starvation of some philosophers (though not all at once), depending on the underlying implementation. In order to conservatively enforce coatomicity, an implementation may prevent an acquire operation from seeing a release operation originating outside its atomic execution, causing the acquire to block forever. So, this is a lock-free implementation, but not a wait-free one. This problem is discussed further in Section 9.3.

5.3.4 Atomic Timeout

One of Orc's most useful idioms is timeout: if an expression executes for longer than a given time period, kill it. We would like to find an analogous approach to **atomic** expressions: after a given time period, force an abort and try again.

Consider an expression `f`, which we would like to run atomically. For simplicity, we will assume that `f` publishes exactly once, and that the published value is irrelevant. We would like to limit `f`'s execution time to one second; if it runs longer, it will be aborted.

The simplest approach is to enforce timeout with pruning, just as with any Orc expression, since killing an **atomic** expression is equivalent to aborting it:

```
val x = atomic f | Rwait(1000)
```

However, this approach has a significant drawback: if the timeout occurs, and **`atomic`** `f` is killed, then it is never retried. How might we cause the timeout to retry `f`? We can use recursion:

```
def attempt() =
  val b = (atomic f) >> true | Rwait(1000) >> false
  if b then signal else attempt()
attempt()
```

If **`atomic`** `f` completes, then `b` is set to `true`, and the expression publishes **`signal`**. If the `Rwait` call publishes first, then `b` is instead set to `false`, and `attempt` is called recursively, retrying the execution of **`atomic`** `f`.

However, this program is incorrect. If **`atomic`** `f` completes, but `Rwait` also simultaneously returns, then `false` could be published, causing a retry of **`atomic`** `f` even though it succeeded. In fact, an arbitrary number of successful executions of **`atomic`** `f` are then possible. Unless `f` is idempotent, this is clearly not satisfactory.

How do we address this? The problem is very similar to the guarded choice example in Section 5.2. We can use ++ to correct the error:

```
def attempt() =
  val b =  (f >> true)
        ++ (Rwait(1000) >> false)
  if b then signal else attempt()
attempt()
```

Notice that **atomic** no longer occurs explicitly in the program; it is implicit in the use of ++.

This encoding assumes that f publishes exactly once. We can relax this assumption by using a more complex encoding to redirect the publications of f. The following program stores the publications of f in a buffer, and then publishes the contents of that buffer if f completes:

```
def repeat(f) = f() >x> (x | repeat(f))
val buffer = Channel()

def attempt() =
  val b =  (f >x> buffer.put(x) >> stop ; true)
        ++ (Rwait(1000) >> false)
  if b then
    repeat(buffer.getD)
  else
    attempt()

attempt()
```

As f runs (as an atom, due to ++), its publications are stored in buffer. If it completes, b is set to true, and the modifications to the buffer are made visible. Then the call repeat(buffer.getD) repeatedly publishes values from the buffer until it is empty, and then halts. If Rwait publishes first, then b is instead set to false and f is killed; the recursive call attempt() retries f.

Note that repeated retries of f use the same buffer, but since the buffer appears empty to all outside observers until f succeeds, there is no risk of corrupting the state of the buffer.

5.3.5 Retry Tactics

So far, we have been using the unary form of the atomic combinator almost exclusively. While this simple approach suffices for most uses of **atomic**, it is sometimes helpful to have more sophisticated retry tactics. Here are a few examples.

Limited Retry

Suppose we would like to execute the expression f atomically, but allow only a limited number of retries before giving up and halting. Here is one approach:

```
def try(n) = Ift(n :> 0) >> atomic f try(n-1)
```

A call to try(n) tries to run f atomically, at most n times. If execution of f aborts, then try(n-1) is called, retrying the execution. However, if there are no retries remaining, the condition n :> 0 fails to hold, and the call simply halts.

Timed Retry

Suppose we would like to execute the expression f atomically, with unlimited retries, but not retry more often than once every two seconds. Here is one approach:

```
def try() =
  val x = Rwait(2000)
  atomic f (x >> try())
try()
```

Execution of f in **atomic** begins in parallel with the call Rwait(2000). If f aborts, but the Rwait call has not yet returned, then x will remain unbound. Once Rwait returns, x is bound, and the recursive call try() occurs, retrying the execution of f.

Limited Timed Retry

We can combine limited retry with timed retry, to allow a limited number of retries with a minimum time interval between each attempt:

```
def try(n) =
  val x = Rwait(2000)
  Ift(n :> 0) >> atomic f (x >> try(n-1))
```

Exponential Backoff

Suppose we would like to execute the expression f atomically, with unlimited retries, but increase the delay between each retry. In particular, we would like to use binary exponential backoff, as in the Ethernet protocol.[Lam13] This tactic is particularly useful when resources are heavily contended.

```
def try(k) =
  atomic f (Rwait(random(2^k + 1)) >> try(k+1))
try(0)
```

The try function is called with an initial argument of 0. If execution of f aborts, then try is called recursively, retrying the execution, but only after a random delay of no more than 2^k milliseconds.

5.3.6 Job Priority

Consider the problem of scheduling jobs with different priorities. We program a scheduler which always executes the highest-priority available job. If a higher priority job arrives while a lower priority job is executing, the executing job is preempted, and the higher priority job runs instead. In other words, a low priority job should never complete while a high-priority job is waiting.

A list of channels, cs, contains the currently available jobs. Each job is an Ora function with no arguments; the job is considered complete when the function call halts. The channel order in cs gives the priority of a channel; the channel at the head of the list contains the highest priority jobs. The function prio(cs) completes the highest priority job available in cs, and then halts. The function exec repeatedly calls prio to execute all available jobs.

```
def block() = Cell()?
def noop() = stop

def prio([])   = block()
def prio(c:cs) =
  val job = c.get() ++ (prio(cs) ; noop)
  job() >> stop

def exec(cs) = prio(cs) ; exec(cs)
```

The prio function traverses the list cs, initiating nested atomic executions on each recursion, since ++ runs each of its branches atomically. If a job arrives from c.get(), it is bound to job, and the search for (and partial execution of) lower-priority jobs is discarded without effect. job is then executed. If prio(cs) completes first, it will publish noop, causing job() to halt without doing anything.

The job of interest has already been found deeper in the list and executed to completion, so nothing needs to be done at this level. The base case for prio is an empty list, where it simply blocks (the block function creates an empty Cell and then immediately reads it, causing it to block forever).

Chapter 6

Implementing Ora

The previous chapter introduced the language Ora, and its central feature, the **atomic** combinator. In this chapter we will see how **atomic** is implemented, specifically how the Ora program cooperates with the sites that it invokes to maintain atomicity, coatomicity, and consistency.

Each time an **atomic** expression executes, a dynamic object called a *transaction* is created to mediate the interaction between that execution and the sites that it calls. This mediation is implemented by two versioning algorithms. The *logical versioning algorithm* manages the state that sites observe from other transactions, ensuring coatomicity. The *resource versioning algorithm* manages the reintegration of state changes that were made within a transaction, ensuring consistency. Atomicity is ensured by a synchronized *commit* operation, together with the two versioning algorithms.

Many of the concepts used in this chapter — transactions, commitment, and so on — are familiar from the broader transactional memory literature. The algorithms shown here are based on a particular strategy for implementing transactions called multiversion concurrency control, or MVCC [BG83]. They were partially inspired by the concepts presented in JVSTM [CRS06], an extension of the Java programming language with transactional memory capabilities. This implementation extends those concepts to account for the pervasive interaction between transactions and orchestrations: transactions can be nested, can run in parallel with each other inside of other transactions, and can interact with concurrent activities occurring in enclosing transactions or even outside of any transaction. The algorithm that maintains coatomicity also incorporates many concepts from the Chandy-Lamport distributed snapshot algorithm [CL85]. The implementation is, in a sense, a hybrid of multiversion concurrency control and distributed snapshot.

As in previous chapters, this is an informal presentation. A formal version of these algorithms is included in the formal semantics of Ora, in Chapter 7.

6.1 Transactions

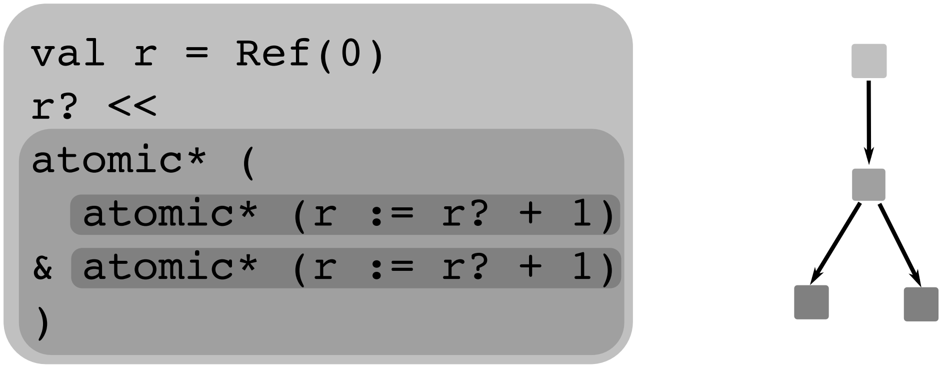
When an atomic expression begins to execute, a transaction is created for that execution. A transaction is a dynamically instantiated object that tracks the events that occur during the execution, to ensure that they satisfy the properties of atomicity, coatomicity, and consistency.

Since the body of an **atomic** expression can itself be any Ora expression, there could be concurrent activity within an atomic execution, due to uses of the parallel combinator or pruning combinator. Furthermore, **atomic** itself might occur in the body expression, and so atomic executions may be nested; any number of concurrent transactions could occur within the context of another transaction. Thus, transactions form a tree, where each non-root node is associated with a transaction, and any transaction that begins within another transaction becomes its child in the tree. Every event of the program is located at a particular node in the tree. Events that do not occur within any atomic expression are located at the root. Figure 6.1 shows an example Ora program, and the transaction tree generated by its execution.

Within a recursive function, the same syntactic occurrence of **atomic** might be executed multiple times. Each of these executions has its own unique transaction. Figure 6.2 shows the recursive job priority function introduced in Section 5.3.6, and a depiction of the (potentially unbounded) transaction tree created during its execution. The function creates new transactions on each recursive call, due to the use of the atomic choice (++) operator. The resulting transaction tree is a right spine of nested transactions.

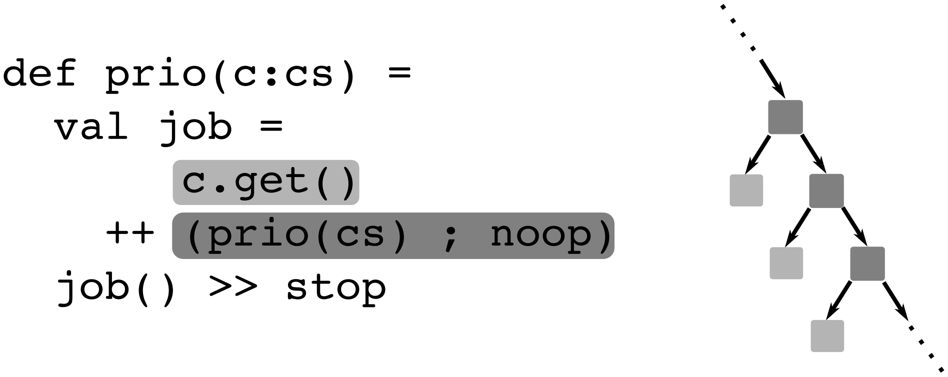
6.1.1 Transactional Sites

Site calls are the primary locus for the implementation of transactions. Each time a site call occurs, the call reports its *location* — the transaction from which the call originated. The site may use this location information to ensure that the call



```
val r = Ref(0)
r? <<
atomic* (
  atomic* (r := r? + 1)
& atomic* (r := r? + 1)
)
```

Figure 6.1: Transaction tree for two increments



```
def prio(c:cs) =
  val job =
        c.get()
     ++ (prio(cs) ; noop)
  job() >> stop
```

Figure 6.2: Transaction tree for job priority function

complies with atomicity, coatomicity, and consistency. Sites fall into three broad categories in terms of their interaction with transactions:

1. Pure sites (as defined in Section 2.3.4) perform stateless computations; they have no causal relationship with other events in the program. Consequently, they cannot violate atomicity, coatomicity, or consistency. They ignore transactions entirely, exhibiting the same behavior regardless of the call's location.

2. Some sites provide access to shared state, such as mutable references or channels. In particular, the sites created by `Ref`, `Cell`, `Guard`, `Channel`, and `Semaphore` fall into this category. These sites do interact with transactions, and that interaction will be the focus of the remainder of this chapter.

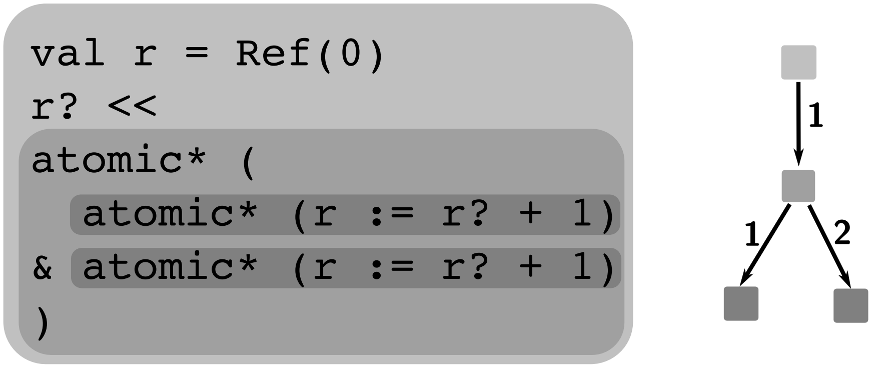
3. Many other sites are available in Ora programs, but instrumenting them to correctly obey atomicity and coatomicity is beyond the scope of this work, due to the complexity or impossibility of defining their causal relationships. So, these sites ignore transactions entirely. One such site is `Rwait`, which is difficult to instrument due to the interaction of causality and temporal order; this problem is discussed further in Section 9.2. Other sites such as `Print` and `Prompt` interact with human users, so it is not even clear how to define causality for them.

6.2 Observation

In order to ensure the property of coatomicity, every site call located in a transaction needs to operate with the same view of the events outside of that transaction. This section describes the algorithm that accomplishes this goal.

Effectively, each transaction operates on a snapshot of the state of each resource with respect to the operations occurring on that resource outside the transaction, to ensure that each event within the transaction sees the same set of outside events and thus ensure coatomicity.

The naive approach to maintaining such a snapshot would be to interpret the intuition of coatomicity literally, recording the state of all resources at the point when the transaction begins. However, this is terribly inefficient, since any given transaction will likely access only a tiny fraction of the full set of resources in the program. Instead, the algorithm creates the snapshot on demand.



```
val r = Ref(0)
r? <<
atomic* (
  atomic* (r := r? + 1)
& atomic* (r := r? + 1)
)
```

Figure 6.3: Transaction tree with logical versions

Whenever an operation inside the transaction makes its first query about the state of a resource outside of the transaction, that query is noted in such a way that each subsequent query will return the same state — even if the resource is later modified. This is ensured by maintaining snapshots on a per-resource basis. Whenever a resource is modified, that modification must check if any such queries have been made, and save the queried state in a log, to ensure that subsequent queries see only that saved state, and not the modifications subsequent to the query.

6.2.1 Version Information

We augment the transaction tree described in the previous section by labeling each edge with a natural number, called a *logical version*. When a new transaction begins, a new node is created, and the edge from its parent is assigned a new logical version, equal to the maximum logical version of its siblings plus 1. This is called the *initial version* of the transaction. If there are no siblings, the initial version is set to 1. Figure 6.3 repeats the example from Figure 6.1 with initial versions for each edge of the transaction tree. Note that the initial version assigned to a child is used only in the context of its parent, so initial versions need only be unique among siblings.[1]

Each shared resource that supports transactional access, such as a reference created by `Ref` or a channel created by `Channel`, has a corresponding *resource tree*.

[1] As a consequence, in a distributed implementation, the parent transaction can provide the initial versions for its children; no global counter is needed for initial versions.

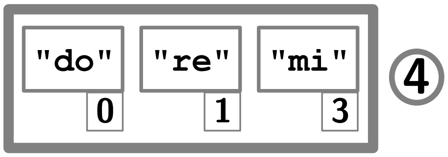


Figure 6.4: Example of a resource node

A resource tree has the same structure as the transaction tree, where each node of the tree — called a *resource node* — contains a sequence of states for that resource. Each state is indexed with a logical version. The sequence is initially empty. Each resource node also has another logical version, called a *boundary number*, not associated with any state. The boundary number is initially 0.

Figure 6.4 gives an example of a resource node, with a sequence of three states and a boundary number of 4. Note that logical versions are always in increasing order, but might not be contiguous; in this example, the logical version 2 has been skipped.

6.2.2 Observing States

Each entry in the sequence of states at a resource node represents the boundary of a snapshot of that resource. Logical versions identify the snapshots. The boundary number identifies the logical version of the next snapshot to be taken; if it is equal to the current maximum logical version, no snapshot is currently needed.

When a site call interacts with a resource, the *observed state* of the resource is determined as follows:

1. Examine the resource tree, at the node corresponding to the call's location. Consider the sequence of states at that node.

2. If the sequence is not empty, the observed state is the state in the sequence with the largest logical version.

3. If the sequence is empty, note the initial version n on the edge from this node to its parent.[2]

[2] Note that the algorithm will never reach this step for the root node; any call that can access

4. If the boundary number of the parent node is less than n, increase it to n.
5. Then examine the sequence at the parent node, but consider only the subsequence with logical versions less than n. Using this subsequence, repeat from step 2.

Figures 6.5 - 6.8 show a few examples of observing a state in a `Ref` resource tree. The transaction tree is shown on the left, and the corresponding nodes of the resource tree are shown on the right. The observed state is indicated by the $\star$ marker. Unseen states (i.e., the states disregarded by step 5 above) are shaded gray.

a resource can also observe, somewhere along the path to the root, the initial state established by the site that created the resource.

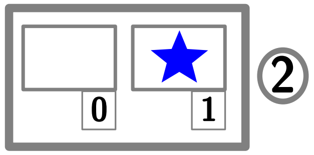


Figure 6.5: Observed state is within caller's transaction

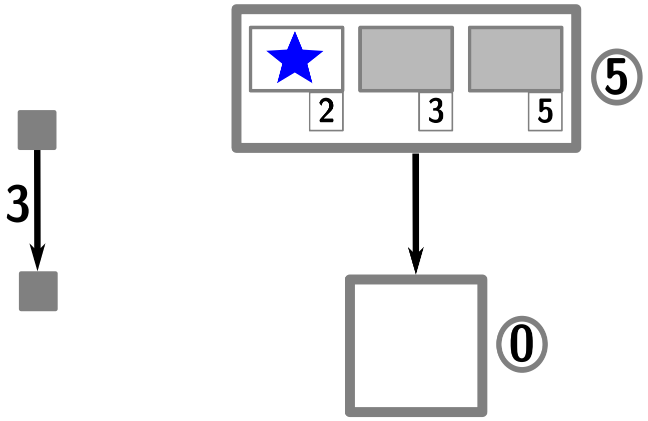


Figure 6.6: Observed state is in parent transaction

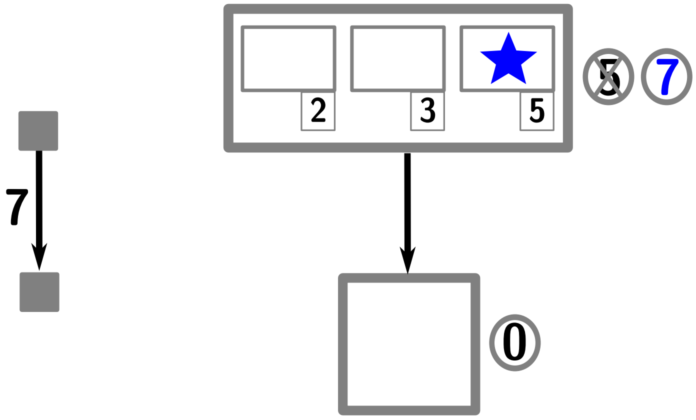


Figure 6.7: Observed state is in parent transaction, boundary number increases

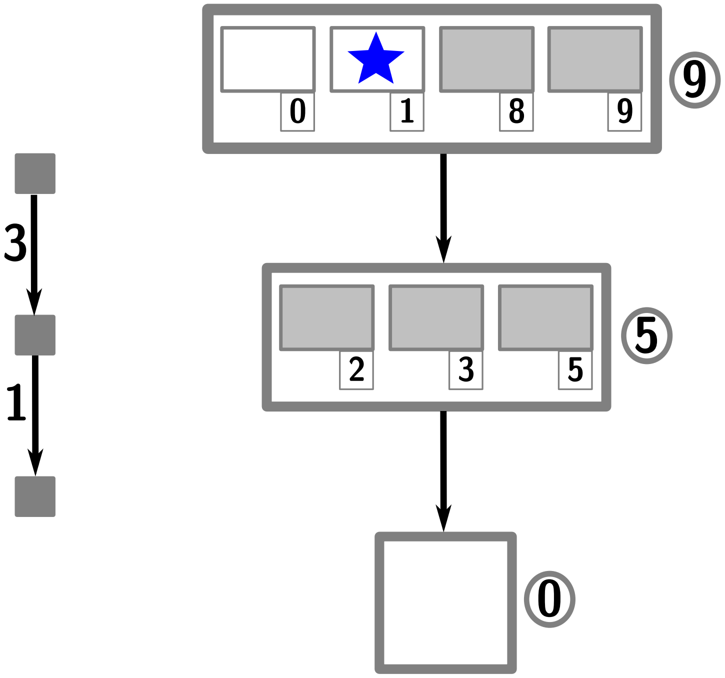


Figure 6.8: Observed state is in parent's parent transaction

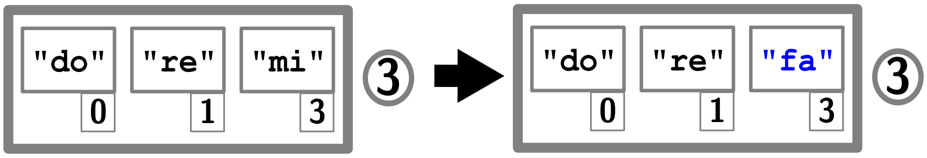


Figure 6.9: When boundary number equals max version, overwrite

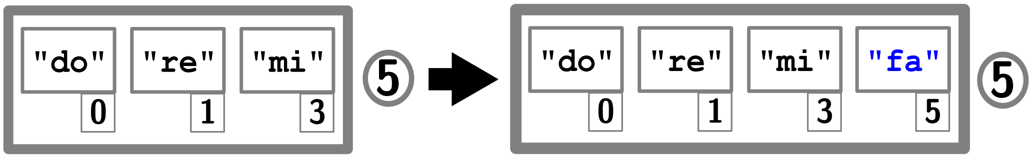


Figure 6.10: When boundary number exceeds max version, append

The site call then performs its computation using just the observed state. If this computation modifies the state, the modified state is added to the resource node for the call's location, with a logical version equal to the boundary number of that resource node. Any previous state with that same logical version is overwritten. Figures 6.9 and 6.10 demonstrate the two possible outcomes.

6.2.3 Blocked Calls

It is possible that the observed state would cause the site call to block; for example, if the operation is a **get** and the observed state is an empty channel. In this case the call must wait until the observed state changes in such a way as to unblock the operation. Each node for a resource with blocking operations (such as a channel or semaphore) has a *waiting set* containing blocked calls. When computing the observed state, at each node where the observed state would cause a call to block, whenever a boundary number would be increased (as indicated in step 4), instead the call is added to the waiting set for that resource node, and its entry in the set is labeled with the logical version that it would have assigned as the new boundary number. If the boundary number would not be increased, no entry is added to the queue. A blocking call is always added to the waiting set for the node of its own transaction, with no label.

Whenever the boundary number of a node is advanced, all calls with a version label less than or equal to the new boundary number are purged from the waiting set. Whenever a state is changed in such a way that it could unblock some calls, the calls to be unblocked are chosen from the waiting set (after it has been purged). The new state is assigned a logical version no less than the maximum of all labels for all calls unblocked in this way.

This subtle relationship between transactions and blocking site calls can give rise to some surprising program behaviors; these are discussed in greater detail in Section 9.3.

6.2.4 Tracking Causality

In order for logical versions to correctly enforce coatomicity, one more refinement is needed: the algorithm must account for the causal relationships induced by the Ora program itself.

Consider the following program:

```
val x = Ref(0)
val y = Cell()

  atomic (x?, y? >> x?)
|
  x := 5 >> y := signal >> stop
```

This program performs two activities in parallel. Within an atomic execution, it reads the value of reference `x`, and in parallel waits for the cell `y` to be assigned and then reads `x` again, publishing a tuple of the two results of reading `x`. Outside of the atomic execution, the program sets `x` to `5`, and then sets `y` to **`signal`**.

When the atomic execution begins, a transaction is created and added to the transaction tree; the edge to the new transaction has a logical version of 1. Subsequently, either `x := 5` or `x?` could occur first.

- If `x := 5` occurs first, then `x?` will observe the state of the reference to be `5`. Subsequently, `y := `**`signal`** occurs, allowing `y?` to proceed, so the second `x?` operation occurs, also observing `5`. The resulting publication in this case is `(5,5)`.

- If `x?` occurs first, then `x?` will observe the state of the reference to be `0`, its initial state. It also advances the boundary number of its parent node in the resource tree of `x`; the new boundary number is 1. Subsequently, `x := 5` occurs, creating a new state in that parent node, with value `5` and logical version 1. Then `y := **signal**` occurs, allowing `y?` to proceed, so the second `x?` operation occurs. That operation observes `0`; since the newly added state of `5` has logical version 1, it is not seen. The resulting publication in this case is `(0,0)`.

There is a problem here: the result `(0,0)` is not actually valid! `x := 5` causally precedes `y := **signal**`, due to the use of the >> combinator. `y := **signal**` causally precedes `y?`, since the read cannot proceed until the cell is assigned. `y?` precedes one of the `x?` operations, again due to the >> combinator. Since causality is transitive, `x := 5` thus causally precedes the `x?` operation. Thus by the definition of a read on a reference, `x?` must publish `5`; it cannot possibly publish `0`. Coatomicity further requires that the causal precedence be shared with the other `x?` operation, so it must also publish `5`.

The logical versioning algorithm has not correctly accounted for the causality within the Ora program. Suppose we extend the algorithm in the following way:

- Whenever a state is observed by a site call, if the state was observed in the same transaction as the call occurred, the site attaches the logical version of that state to the call's response.
- Whenever a site is called in a transaction, it carries the maximum logical version of all of its causes from within that same transaction, such as site responses. The mechanism that conveys these logical versions through the Ora program is presented in detail in Section 7.2.
- Whenever a site call modifies a state, the version attached to that state is the maximum of the boundary number and the version attached to the call.

With this augmentation, we reexamine the scenario where the incorrect publication `(0,0)` occurred.

Suppose `x?` occurs before `x := 5`. Then `x?` will observe the state of the reference to be `0`, its initial state. It also advances the boundary number of its parent (the root) in the resource tree of `x`; the new boundary number is 1. Subsequently, `x := 5` occurs, creating a new state in the root of `x`'s resource tree, with value `5` and logical version 1. The call `x := 5` responds with **`signal`**, and that response now carries a version of 1. Subsequently `y :=` **`signal`** occurs; the call carries a version of 1 since it was caused by `x := 5`. This adds a new state to the root of the resource tree of `y`, with value **`signal`** and version 1. Now, when `y?` occurs, it blocks indefinitely; it does *not* see the assignment of **`signal`** to `y`, since that state has logical version 1, which is not less than the edge's logical version of 1. Consequently, the transaction blocks, forever. An incorrect result has been prevented, at the cost of progress; this tradeoff is discussed in greater detail in Section 9.3.

6.2.5 Relationship to Distributed Snapshot

This logical versioning algorithm incorporates elements of the distributed snapshot algorithm developed by Chandy and Lamport [CL85]. In particular, the tracking of causality through the Ora program is the key to the relationship between the two algorithms. The comparison that follows is based on Dijkstra's presentation of the algorithm [Dij83].

The snapshot algorithm operates over a strongly connected network of machines, which communicate only by messages sent through ordered, first-in-first-out input buffers. The machines are analogous to the set of shared resources in Ora, together with the set of all threads of execution within the Ora program. In the distributed snapshot algorithm, the intent is to capture a global state of the entire system. In Ora, we are interested in only a subset of such a global snapshot: we need only a snapshot of the shared resources observed by a particular transaction.

The "red letter" — the unique message that computes the boundary of the distributed snapshot — corresponds to the initial version number assigned to a new transaction. As the transaction proceeds, it observes various resources, and as it does so, it affects the boundary numbers of those resources by reading its own initial version and advancing the boundary number to at least that version. Each read of the initial version corresponds to the transmission of the red letter from the "starter" machine in the distributed snapshot, and the advancement of a resource's boundary number corresponds to receipt of the red letter from the starter machine. Then, the manipulation of the resource node to ensure that further state changes do not overwrite the existing entry corresponds to the local state recording performed by a machine as it changes from white to red. In this way, the global state computed by the distributed snapshot corresponds exactly to the set of events seen by all members of the transaction as required by coatomicity.

The correctness of the distributed snapshot algorithm relies on the fact that each machine, upon first receiving a red letter, sends a red letter to each other machine. Since the transmission of messages from machine to machine could be delayed, it is possible that a machine receives its first red letter from a machine other than the starter, in which case it will already have turned red by the time the red letter from the starter arrives. Similarly, the logical versioning algorithm relies on the correct transmission of logical versions through causal relationships in the Ora program, and through sites; if the boundary version of a resource node has been advanced by an Ora operation before the resource is observed by a child transaction, this is similar to the receipt of a red letter by a machine already colored red.

Note that Ora's logical versioning algorithm must handle many simultaneously active sibling transactions, with different initial versions. The situation is analogous to a cascading execution of many distributed snapshots. However, the situation is easily managed, since the version numbers are totally ordered.

6.3 Merging

The logical version algorithm described in the previous section enforces coatomicity. However, it does not enforce atomicity, since it does not describe how parts of the program outside of a transaction can observe events that occurred within that transaction. When a transaction is finished, its state changes must be moved into its parent so that they become visible. This merging process is additionally constrained by the requirements of consistency, as defined in Chapter 4; once the changes have been made visible, each observer that could see those changes must still see a unique state for each resource.

In this section we will see how and when a resource tree node is combined with its parent node, through a *commit* process that checks whether each resource can be combined in a way that preserves consistency. This algorithm requires that resource states be augmented with additional elements, called *resource versions*, which track the operations that have been performed on the resource.

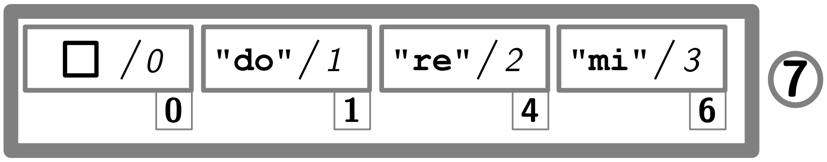


Figure 6.11: A Ref resource node, augmented with resource versions

6.3.1 Resource Versions

A *resource version* is a new element of a resource node entry; it is a natural number that counts how many times a particular operation has been performed on that resource. Resource versions summarize the history of state changes on a resource in such a way that consistency can be ensured simply by comparing resource versions. Whenever a state is modified, its resource versions are modified in tandem. Here, we describe the state and version information maintained by the resource node entries of each resource type.

Guard

A Guard entry has a state of **true** if the guard has been claimed, **false** otherwise. It has no resource versions; its state carries sufficient information.

Ref and Cell

A Ref or Cell entry has a single optional value as its state, and one resource version. The resource version counts the number of times the reference has been written; in the case of cells, only 0 or 1 is possible. Figure 6.11 gives an example of a resource tree node with four entries, each of which has a state (the first state is □, indicating an empty reference), and a resource version.

Channel

A Channel entry contains a sequence of values as its state, and two resource versions, g and p. The version g counts the total number of `get` operations that have been performed on the channel; similarly, p counts the total number of `put` operations.

`Semaphore`

A `Semaphore` entry has two resource versions, a and r, and no state. The version a counts the total number of `acquire` operations that have ever been performed on the semaphore; similarly, r counts the total number of `release` operations. Whenever the semaphore count is needed, it can be calculated directly as $(r - a)$. An `acquire` operation may only proceed if $r > a$.

6.3.2 Commit

When the body expression of an atomic expression halts, the corresponding transaction attempts to *commit*. The commit process combines the current state of each resource in the transaction with the current state of that resource in the transaction's parent, while ensuring that consistency has been maintained. If consistency has not been maintained, then there is a *conflict*, and an abort occurs, causing the alternate branch of the **atomic** expression to be evaluated instead, as described in Section 5.1. In the case of a conflict, no changes are made to any states in the transaction's parent.

For each resource tree, if the committing transaction's node in the tree has a non-empty sequence of entries, then that resource tree is a *participant* in the transaction. Each participant identifies three resource node entries in its resource tree:

- The current child entry, q_{child}. This is the entry with the largest logical version in the committing transaction's resource node.
- The current parent entry, q_{parent}. This is the entry with the largest logical version in the committing transaction's parent's resource node.
- The branch entry, q_{branch}. This is the entry that would be observed by the committing transaction if its own resource node contained an empty sequence. Let q_{branch} be its resource version.

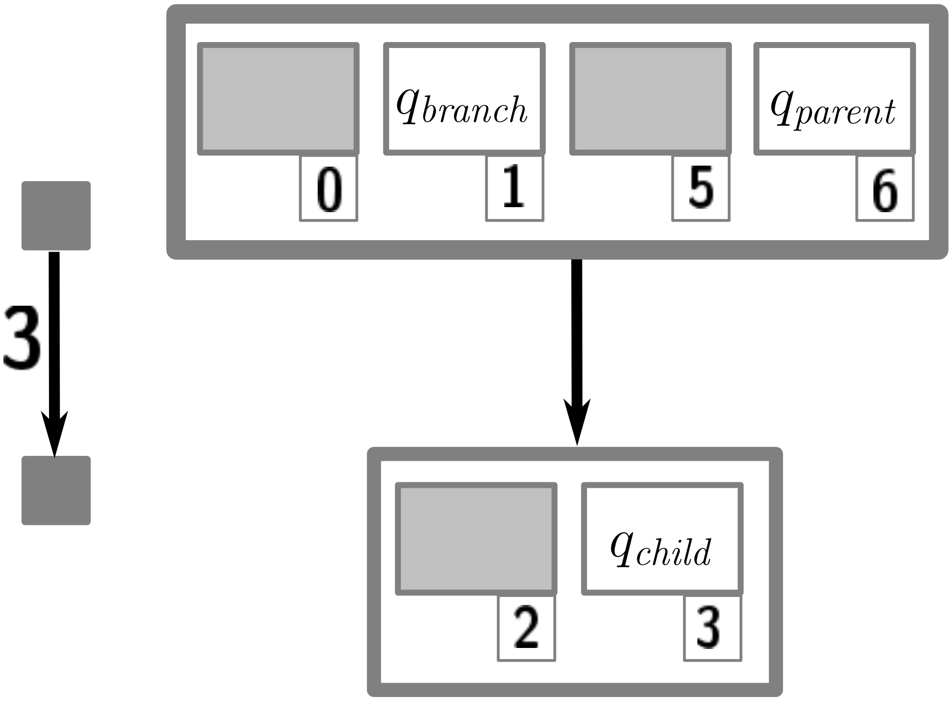


Figure 6.12: Selection of child, parent, and branch entries

Figure 6.12 shows an example of this selection process.

With these entries identified, each participant attempts to merge its own q_{child} and q_{parent} to create a new q_{merge} entry. In all cases, a merge will succeed if q_{parent} and q_{branch} are equal[3]; the resulting merged entry q_{merge} is just q_{child}.

The merging procedure is different for each resource:

`Guard`

The merge succeeds if only one of q_{parent} or q_{child} is **true**. If so, the merged entry is **true**. Otherwise, the merge fails.

`Ref` and `Cell`

The merge succeeds only if one of q_{parent} or q_{child} has a different resource version than q_{branch}, but not both. If so, the merged entry is equal to whichever of q_{parent} or q_{child} had the differing resource version. This ensures that divergent writes always produce a conflict.

`Channel`

The merge succeeds under two conditions:

- q_{branch} has the same count of **put** operations as q_{child}, or q_{parent}, or both.
- q_{branch} has the same count of **get** operations as q_{child}, or q_{parent}, or both.

These conditions ensure that **put** operations do not occur in both the parent and the child, nor do **get** operations. If **put** operations occurred in both the parent and the child, then they would need to be ordered in the channel, and since both orderings are permissible, multiple states are possible, violating consistency. If **get** operations occurred in both the parent and the child, then duplicates of the same channel items have been used by multiple computations, violating the expected behavior of a channel.

[3] Note that the merge will also always succeed if q_{child} and q_{branch} are equal, but in that case, the resource would not even be in the set of participants to begin with.

The merged entry q_{merge} for the channel has the larger of the **put** counts from q_{child} and q_{parent}, and the larger of the **get** counts from q_{child} and q_{parent}. Its state is the sequence of values from the entry with more **put** operations; if that entry did not also have more **get** operations, then one element is removed from the head of the sequence for every extra **get** operation in the other entry.

`Semaphore`

The merge succeeds if the sum of the **acquire** counts in q_{parent} and q_{child} does not exceed the sum of the **release** counts in q_{parent} and q_{child}. This condition ensures that the merged semaphore will be in a legal state.

The merged entry q_{merge} has the total of the **acquire** counts from q_{parent} and q_{child}, minus the **acquire** count of q_{branch}, and a **release** count calculated in the same way.

If any participant fails to merge, then there is a conflict, and the transaction is aborted.[4] If all participants successfully merge, then the maximum logical version n of all parent entries is calculated; call this the *commit version*. Then, each parent node is updated with a new entry, q_{merge}, with logical version n. Additionally, each publication in the Ora program that was unfrozen by the commit is given a logical version of n, so that any events that those publications cause will have a logical version of at least n.

In order to ensure atomicity, all of the updates to the participants must occur simultaneously, so that if an observer in the parent sees one modification resulting from the commit, it sees all of the modifications. This can be ensured by any reasonable commit protocol, such as a two-phase commit [BN09]. Many commit protocols will require some way to prioritize transactions when a conflict occurs; in those cases, the initial version of the transaction can be used as a tiebreaker.

[4] Note that if the committing transaction did not modify any states, then the set of participants is empty, so conflict is impossible. Thus, Ora shares an important property with other multiversioning transaction systems: a read-only transaction will not abort.

Chapter 7

Formal Semantics of Ora

This chapter describes the formal grammar and small-step operational semantics of Ora. At its core, the formal semantics is an extended version of the Orc formal semantics introduced in Section 2.3. It formalizes the language extensions introduced in Chapter 5 and the transactional implementation of those extensions described in Chapter 6. This formal representation is necessary to provide a firm mathematical foundation for the formal proofs of atomicity, coatomicity, and consistency presented in Chapter 8.

We begin with the grammar of Ora programs, which extends the Orc grammar with support for **atomic**, and also modifies many expressions to carry logical version information. Subsequently we consider the *internal semantics* of Ora, which describes the labeled transition relation for Ora programs. This includes the transitions originally seen in the Orc semantics — with extensive modifications to transport logical version information through the program — as well as new transitions to support the execution of **atomic**. Then we see the *external semantics* of Ora, which describes the structure of the *environment* in which an Ora program runs, and the small-step transitions which the program and its environment jointly make as they interact.

The algorithms described in Chapter 6 are formally modeled by the *snapshot semantics*, which controls the observation of states, and the *commit semantics*, which describes the conditions under which a transaction may commit or abort. Each of these relies on a small group of supporting operations, which manipulate sets of events in the environment. Note that the data structures described in Chapter 6 capture only the subset of the environment needed for the algorithm, whereas the formal semantics captures the entire history of the environment, including items such as overwritten states or boundary numbers.

The chapter concludes with a description of the *resource semantics*, which formally models the computation that occurs at sites and the resource-specific operations on which the other semantic rules rely.

7.1 Syntax of Ora

Figure 7.1 defines the formal syntax of Ora programs.

Variables and handles are arbitrary identifiers. All variable names occur in the initial program, whereas handles are generated as the program runs. The set of values contains all sites, all constants, all structured values, and all other datatypes returned by sites. Each of these concepts is the same as for Orc programs.

Ora introduces two new formal syntactic entities: transactions (t) and logical versions (n). Transactions are arbitrary identifiers, generated at runtime, just like handles. Logical versions are natural numbers; some syntactic entities are augmented with logical versions.

The syntactic definitions of values (v), responses (w), and declared functions ($\textbf{def}\ y(\bar{x}) = f$) are unchanged. Parameters (p) may be either variable names or responses, but whenever a parameter is a response, it now carries a logical version.

The syntax for Ora expressions includes some Orc expressions unchanged, some with changes, and some entirely new syntactic cases. The syntax of standalone parameters (p) and site calls ($p(\bar{p})$) remains unchanged, except that the syntax for parameters itself has changed as described above. The syntax for calls in progress ($k?$), and for the four Orc combinators, remains unchanged. The syntax for declaring a function in a scope now includes a logical version n attached to the function D.

Ora adds three new expressions: versioned expressions (f^n), **atomic**, and **trans**. A versioned expression simply carries a logical version. The expression **atomic** $f\ g$ is an instance of the **atomic** combinator that has not yet begun to execute; f is the body expression and g is the fallback expression, as described in Section 5.1. The expression **trans** $t\ f\ (h, g)$ is an active execution of an **atomic** expression. The transaction t is its unique identifier, f is the executing body expression, g is the fallback expression, and h is a special expression that contains all values published by f so far; when the transaction commits, h will be executed.

If a program has not yet begun to execute, all logical versions n must be 0, and the expressions $k?$, f^n, and $\textbf{trans}\ t\ f\ (h, g)$ cannot occur in the program.

$$
\begin{array}{rcllcl}
x, y & \in & \mathit{Variable} & & & \\
k & \in & \mathit{Handle} & & & \\
V & \in & \mathit{Value} & & & \\
 & & & & & \\
s, t, u & \in & \mathit{Transaction} & & & \\
n & \in & \mathbb{N} & & & \\
 & & & & & \\
v & \in & \mathit{Orc\ value} & ::= & & V \mid D \\
w & \in & \mathit{Response} & ::= & & v \mid \textbf{stop} \\
p & \in & \mathit{Parameter} & ::= & & w^n \mid x \\
 & & & & & \\
D & \in & \mathit{Definition} & ::= & & \textbf{def}\ y(\bar{x}) = f \\
 & & & & & \\
f, g, h & \in & \mathit{Expression} & ::= & & p \\
 & & & & \mid & p(\bar{p}) \\
 & & & & \mid & k? \\
 & & & & \mid & f >x> g \\
 & & & & \mid & f \mid g \\
 & & & & \mid & f <x< g \\
 & & & & \mid & f\ ;\ g \\
 & & & & \mid & (D^n)\ \#\ f \\
 & & & & \mid & f^n \\
 & & & & \mid & \textbf{atomic}\ f\ g \\
 & & & & \mid & \textbf{trans}\ t\ f\ (g, h)
\end{array}
$$

Figure 7.1: Formal Syntax of Ora

7.2 Internal Semantics

The *internal semantics* of Ora is a small-step operational semantics with labeled transitions. The rules of the internal semantics describe how Ora programs execute; they do not describe the computation of sites, or the relationship between the Ora program and the sites it calls.

These rules define two characteristics of expressions: which transition steps are available for an expression (the *execution judgment*), and whether an expression has halted (the *halting judgment*). These judgments are mutually exclusive; a halted expression has no available transition steps, and an expression with available transition steps has not halted.

7.2.1 Halting Judgment

The halting judgment has the following form, where f is an expression, and n is a logical version:

$$\boxed{f \text{ halted at } n}$$

The judgment states that f has halted, and that n is the maximum logical version among all events that caused f to halt. The judgment is defined inductively by the rules in Figure 7.2:

- The (HaltPar) rule states that a parallel combinator has halted when both of its subexpressions have halted. It halts at the maximum of the versions at which each of its subexpressions halted.

- The (HaltSeq) rule states that a sequential combinator has halted when its left subexpression has halted.

- The (HaltDef) rule states that if an expression consists of a definition and its scope, that expression halts when the scope expression halts.

- The (HaltCall) and (HaltArg) rules indicate the two cases when a call halts: if the value being called becomes **stop**, or if the value being called is a site and one of its arguments is **stop**. In each case the occurrence of **stop** dictates the logical version at which the whole call halts. The choice of

$$\frac{f \text{ halted at } m \qquad g \text{ halted at } n}{f \mid g \text{ halted at } \max(m,n)} \qquad (\textsc{HaltPar})$$

$$\frac{f \text{ halted at } n}{f \mathbin{>x>} g \text{ halted at } n} \qquad (\textsc{HaltSeq})$$

$$\frac{f \text{ halted at } n}{D^m \mathbin{\#} f \text{ halted at } n} \qquad (\textsc{HaltDef})$$

$$\mathbf{stop}^n(\bar{p}) \text{ halted at } n \qquad (\textsc{HaltCall})$$

$$V^m(..., \mathbf{stop}^n, ...) \text{ halted at } n \qquad (\textsc{HaltArg})$$

$$\mathbf{stop}^n \text{ halted at } n \qquad (\textsc{HaltStop})$$

Figure 7.2: Halting Judgment

logical version in (HaltArg) can be nondeterministic if multiple arguments are **stop**.

- The (HaltStop) rule states that **stop** is halted. It halts at the logical version attached to that instance of **stop**.

7.2.2 Execution Judgment

The execution judgment has the following form, where f and f' are expressions, and L is a transition label:

$$\boxed{f \xrightarrow{L} f'}$$

The judgment states that f has a transition step available: it can transition to f', emitting the label L.

Labels

The possible labels L are described by the grammar in Figure 7.3. Some labels refer to transaction paths T; these are either sequences of transaction identifiers t, or the empty sequence ϵ. If a transaction path is written Tt, this simply expresses the requirement that the sequence is not empty.

- A site call label ($V_k(\bar{v})$ in T at n) indicates that the expression is calling the site[1] V with arguments $\bar{v}$. The call originated from transaction path T. The logical version n is the maximum version among all events that caused the call. The call is uniquely identified by the handle k.
- A site return label ($k?w$ at n) indicates that the expression is accepting a return value from a call. The handle k identifies the original call. The response w is either a value v or the negative response **stop**. The logical version n is the maximum version among all events that caused the response, including the original call and possibly including external events.
- A silent label (τ) indicates that the expression is making an internal transition with no effect on the rest of the program, or the environment.
- An init label (**init** Tt) indicates that the expression is starting a new transaction at path Tt.
- A commit label (**commit** Tt at n) indicates that the expression is committing a transaction, and that all of the transaction's events have logical version n.

[1] Note that V is used, not v, to exclude the possibility of a function (D); function calls are treated differently than site calls.

$$
\begin{array}{rclcl}
l & \in & \textit{Non-publication Label} & ::= & V_k(\bar{v}) \text{ in } T \text{ at } n \\
 & & & \mid & k?w \text{ at } n \\
 & & & \mid & \tau \\
 & & & \mid & \textbf{init } Tt \\
 & & & \mid & \textbf{commit } Tt \text{ at } n \\
 & & & \mid & \textbf{abort } Tt \\
L & \in & \textit{Label} & ::= & l \mid\, !v \text{ at } n \\
 & & & & \\
S, T, U & \in & \textit{Transaction Path} & ::= & \epsilon \mid Tt
\end{array}
$$

Figure 7.3: Grammar of Transition Labels

- An abort label (**abort** Tt) indicates that the expression could abort the transaction at path Tt.
- A publication label ($!v$ at n) indicates that the expression is publishing a value v with logical version n.

For convenience, labels that are not publications have a separate syntactic category l, since many rules for the execution judgment discriminate between publication steps and non-publication steps.

Transition Rules

The execution judgment is defined inductively by the rules in Figures 7.4 - 7.9.

Base Expressions Figure 7.4 shows the possible execution steps for *base* expressions, i.e. those that contain no subexpressions.

- Free variables and **stop** are base expressions, but they have no transition rules, since they cannot take any steps; free variables are blocked and **stop** is halted.
- The (SiteCall) rule executes site calls. Execution of a site call creates a new unique handle k to identify the call. The site being called must be a non-function value V; calls to functions are handled by a different rule. Each parameter in $\bar{p}$ must be a value. The call transitions to a handle expression $k?$, emitting a site call label containing the site, the arguments, the handle, an initially empty transaction path, and a logical version n, computed as the max of the logical versions of the site value and the argument values.

$$\frac{\begin{array}{c} k \text{ fresh} \\ p_i = v_i \text{ at } m_i \\ n = \max(m, \max_i(m_i)) \end{array}}{V^m(\bar{p}) \xrightarrow{V_k(\bar{v}) \text{ in } \epsilon \text{ at } n} k?} \qquad (\textsc{SiteCall})$$

$$k? \xrightarrow{k?w \text{ at } n} w^n \qquad (\textsc{SiteReturn})$$

$$v^n \xrightarrow{!v \text{ at } n} \mathbf{stop}^n \qquad (\textsc{Publish})$$

Figure 7.4: Execution Judgment for Base Expressions

- The (SiteReturn) rule processes site responses. A handle expression $k?$ transitions to a response w, which could be a value v, or **stop**. The logical version associated with the response is maintained in the program. The label attached to the transition is a response label, which provides the response value (or **stop**) and identifies the handle receiving the response.[2]

- The (Publish) rule takes a value v with a logical version n, and transitions to **stop** with that same logical version. The transition has a publication label carrying v and n, to be received and used elsewhere in the program.

Functions Figure 7.5 shows the transitions associated with function definitions and function calls.

- The (DefScope) and (DefClose) rules are largely unchanged from the Orc semantics; they allow transitions within a definition's scope, and the substitution of a closed definition into its scope, respectively. The only change is the logical version n attached to the definition, which is carried with the definition as it is substituted.

[2] Technically, transitions for every possible response are available for $k?$ to make; the external semantics (discussed in the next section) chooses one of these possible transitions — the one corresponding to the actual result of the call — when the call has completed.

$$\frac{f \xrightarrow{L} f'}{D^n \;\#\; f \xrightarrow{L} D^n \;\#\; f'} \qquad (\textsc{DefScope})$$

$$\frac{\begin{array}{c} D \text{ is } \mathbf{def}\ y(\bar{x}) = g \\ FV(D) = \emptyset \end{array}}{D^n \;\#\; f \xrightarrow{\tau} [y \mapsto D^n]f} \qquad (\textsc{DefClose})$$

$$\frac{D \text{ is } \mathbf{def}\ y(\bar{x}) = g}{D^n(\bar{p}) \xrightarrow{\tau} [y \mapsto D][\bar{x} \mapsto \bar{p}]g^n} \qquad (\textsc{DefCall})$$

Figure 7.5: Execution Judgment for Functions

- The (DefCall) rule has changed slightly: when a function is called, the logical version of the function is attached to the body expression.

Combinators Figure 7.6 shows the transitions available for each of the four Orc combinators. The rules (ParL), (ParR), (SeqN), (PruneL), (PruneN), (OtherN), and (OtherV) are all equivalent to their counterparts in the Orc semantics. The other rules have changed solely to accommodate the transit of logical versions through the program.

- The (OtherH) rule takes the logical version n at which the left expression f halts, and attaches it to the right expression g, since all events of g depend on the halting of f.
- Similarly, the (SeqV) rule attaches the logical version of the publication from f to the new instance of g.
- The (PruneV) rule attaches the logical version n of the published value v to each place where v is substituted in f; this is because not all events of f depend on v, just the events that block on the variable x.
- The (PruneZ) rule similarly attaches the logical version n at which g halts to each place where **stop** is substituted in f.

$$\frac{f \xrightarrow{L} f'}{f \mid g \xrightarrow{L} f' \mid g} \quad (\textsc{ParL}) \qquad \frac{f \xrightarrow{l} f'}{f \mathbin{>x>} g \xrightarrow{l} f' \mathbin{>x>} g} \quad (\textsc{SeqN})$$

$$\frac{g \xrightarrow{L} g'}{f \mid g \xrightarrow{L} f \mid g'} \quad (\textsc{ParR}) \qquad \frac{f \xrightarrow{!v \text{ at } n} f'}{f \mathbin{>x>} g \xrightarrow{\tau} f' \mathbin{>x>} g \mid [x \mapsto v]g^n} \quad (\textsc{SeqV})$$

$$\frac{f \xrightarrow{l} f'}{f \mathbin{;} g \xrightarrow{l} f' \mathbin{;} g} \quad (\textsc{OtherN}) \qquad \frac{f \xrightarrow{L} f'}{f \mathbin{<x<} g \xrightarrow{L} f' \mathbin{<x<} g} \quad (\textsc{PruneL})$$

$$\frac{f \xrightarrow{!v \text{ at } n} f'}{f \mathbin{;} g \xrightarrow{!v \text{ at } n} f'} \quad (\textsc{OtherV}) \qquad \frac{g \xrightarrow{l} g'}{f \mathbin{<x<} g \xrightarrow{l} f \mathbin{<x<} g'} \quad (\textsc{PruneN})$$

$$\frac{f \text{ halted at } n}{f \mathbin{;} g \xrightarrow{\tau} g^n} \quad (\textsc{OtherH}) \qquad \frac{g \xrightarrow{!v \text{ at } n} g'}{f \mathbin{<x<} g \xrightarrow{\tau} [x \mapsto v^n]f} \quad (\textsc{PruneV})$$

$$\frac{g \text{ halted at } n}{f \mathbin{<x<} g \xrightarrow{\tau} [x \mapsto \mathbf{stop}^n]f} \quad (\textsc{PruneZ})$$

Figure 7.6: Execution Judgment for Orc Combinators

Transactions Figure 7.9 shows the execution steps for the new expression forms added by Ora: versioned expressions (f^n), the atomic combinator (**atomic** f g), and transaction instances (**trans** t f (h, g)).

- The (Versioned) rule states that a versioned expression f^n can make a transition to $(f')^n$ whenever its body expression f could transition to f'. If the transition of f has the label L, then the transition of f^n has the label L^n, which remaps L to have a minimum logical version of n. The definition of this remapping is given in Figure 7.8.

- The (TransBegin) rule initiates a transaction, converting an atomic expression (**atomic** f g) to a transaction instance (**trans** t f $(\mathbf{stop}, g)$). The identifier t uniquely identifies the transaction. The commit expression h is initially **stop**, since the body expression f has not yet published any values.

- The (TransN) rule handles all non-publication transitions of the transaction. Whenever the body expression f makes a transition, with label l, the transaction makes a transition with label $t(l)$, a remapping of l that extends the transaction path of the label by appending t. The definition of this remapping is given in Figure 7.7. As a result of this remapping, a label will have a transaction path with the identifiers of all transactions enclosing its point of origin by the time that the label propagates to the outside of the program.

- The (TransV) rule manages publications. When the body expression f publishes a value, that publication is frozen by adding it to the commit expression h, and the transaction makes a τ transition instead. When h is later executed, the result will be to publish all values that were frozen in this way.

- The (TransCommit) rule commits a transaction. A transaction may attempt to commit only if its body expression f has halted. The commit event contains only the singleton path t, but it will accumulate the entire transaction path as it propagates upward through the program. It transitions to the success branch h after it commits, making all of the frozen publications of the body expression simultaneously available for publication. This transition rule expresses only the effects of a commit within the Ora program; this transition must synchronize with the environment, to ensure that the commit is valid, and to expose the state changes associated with the commit.

$$\begin{array}{rcl} t(\textbf{init}\ T) & = & \textbf{init}\ tT \\ t(\textbf{commit}\ T \text{ at } n) & = & \textbf{commit}\ tT \text{ at } n \\ t(\textbf{abort}\ T) & = & \textbf{abort}\ tT \\ t(V_k(\bar{v}) \text{ in } T \text{ at } n) & = & V_k(\bar{v}) \text{ in } tT \text{ at } n \\ \\ t(L) & = & L \text{ , for all other labels} \end{array}$$

Figure 7.7: Path Remapping in (TransN)

$$\begin{array}{rcl} (!v \text{ at } m)^n & = & !v \text{ at } \max(m, n) \\ (V_k(\bar{v}) \text{ in } \epsilon \text{ at } m)^n & = & V_k(\bar{v}) \text{ in } \epsilon \text{ at } \max(m, n) \\ \\ L^n & = & L \text{ , for all other labels} \end{array}$$

Figure 7.8: Version Remapping in (Versioned)

Note that there is a logical version n attached to a commit label; as with response labels $k?w$, this version n is actually supplied by the environment at commit time; it is the commit version of the transaction, and it is attached to all publications of h, using a versioned expression h^n.

- The (TransAbort) rule aborts a transaction, making a transition to its failure branch g. As with the commit transition, the transition initially has the singleton path t, but will accumulate the rest of the path as it propagates. A transaction's abort transition is available at all times, but as with commit transitions, an abort transition must also synchronize with the environment.

$$\frac{f \xrightarrow{L} f'}{f^n \xrightarrow{L^n} (f')^n} \qquad (\textsc{Versioned})$$

$$\frac{t \text{ fresh}}{\mathbf{atomic}\ f\ g \xrightarrow{\mathbf{init}\ t} \mathbf{trans}\ t\ f\ (\mathbf{stop}, g)} \qquad (\textsc{TransBegin})$$

$$\frac{f \xrightarrow{l} f'}{\mathbf{trans}\ t\ f\ (h, g) \xrightarrow{t(l)} \mathbf{trans}\ t\ f'\ (h, g)} \qquad (\textsc{TransN})$$

$$\frac{f \xrightarrow{!v \text{ at } m} f'}{\mathbf{trans}\ t\ f\ (h, g) \xrightarrow{\tau} \mathbf{trans}\ t\ f'\ (h \mid v, g)} \qquad (\textsc{TransV})$$

$$\frac{f \text{ halted at } m}{\mathbf{trans}\ t\ f\ (h, g) \xrightarrow{\mathbf{commit}\ t \text{ at } n} h^n} \qquad (\textsc{TransCommit})$$

$$\mathbf{trans}\ t\ f\ (h, g) \xrightarrow{\mathbf{abort}\ t} g \qquad (\textsc{TransAbort})$$

Figure 7.9: Execution Judgment for Versioned Expressions and Transactions

$$\begin{array}{rcll}
[x \mapsto w^n](D^m) & = & ([x \mapsto w]D)^{\max(m,n)} & \text{if } x \in \mathrm{FV}(D) \\
 & = & D^m & \text{if } x \notin \mathrm{FV}(D) \\
 & & & \\
[x \mapsto w^n](\textbf{atomic}\ f\ g) & = & (\ \textbf{atomic}\ ([x \mapsto w]f)\ ([x \mapsto w]g)\ )^n & \text{if } x \in \mathrm{FV}(f) \\
 & = & \textbf{atomic}\ f\ ([x \mapsto w^n]g) & \text{if } x \notin \mathrm{FV}(f)
\end{array}$$

Figure 7.10: Special Substitution Rules

Special Substitution Rules

The substitution operation $[x \mapsto p]$ behaves differently in Ora than it does in Orc, in a few special cases. Some program transitions cannot occur until all free variables have been substituted into certain expressions or function definitions. In these cases, the logical version attached to the value being substituted must also be incorporated into the larger expression receiving the substitution, since any events resulting from its evaluation were caused, in a sense, by the decrease in free variables resulting from the substitution. Figure 7.10 shows these special cases: substitutions into a function D or an atomic expression **atomic** f g.

7.3 External Semantics

In addition to the internal transitions of the Ora program itself, there is a set of semantic rules describing the environment in which the program runs, and the interactions between the Orca program and that environment. It is called the *external semantics*.

7.3.1 Environment and Event Grammar

An *environment*, denoted by E, is a set of *events*. As the Ora program runs, it interacts with the environment through site calls. The internal processing of sites happens within the environment, adding new events to the environment corresponding to new states for shared resources. Other events in the environment keep track of the status of transactions.

The grammar of events is shown in Figure 7.11.

A *resource*, denoted by R, is an identifier associated with each stateful object manipulated by sites. For example, a mutable reference is a resource, and could be manipulated by two different sites: one that reads the reference, and another that writes to it.

A *resource state*, denoted by q, is a representation of the state of some resource. Each resource defines the structure of its resource states differently; the grammars of these resource states are given in Section 7.7.

An event, denoted by e, is a structure with three components: its content c, its transaction path T, and its logical version n. Transaction paths and logical versions were defined in the previous section. The meaning of an event depends on its content:

- A call event ($\langle k \triangleright V(\bar{v}), T, n\rangle$) indicates that a call was made to the site V with arguments $\bar{v}$ and handle k. The transaction path T of the event indicates the location of the call.
- A return event ($\langle k \triangleleft w, T, n\rangle$) indicates that the call identified by handle k has completed, and that the site's response is w.
- An init event ($\langle \mathbf{init}\ t, T, n\rangle$) indicates that a new transaction t has begun. The transaction path T indicates the parent of t, and n is the logical version

$$
\begin{array}{rclcl}
R & \in & \mathit{Resource} & & \\
 & & & & \\
e & \in & \mathit{Events} & ::= & \langle c, T, n \rangle \\
c & \in & \mathit{Content} & ::= & k \rhd V(\bar{v}) \\
 & & & \mid & k \lhd w \\
 & & & \mid & \textbf{init } t \\
 & & & \mid & \textbf{commit } t \\
 & & & \mid & \textbf{advance } R \\
 & & & \mid & q_R \\
 & & & & \\
q & \in & \mathit{Resource\ State} & &
\end{array}
$$

Figure 7.11: Grammar of Events

associated with the edge from T to t in the transaction tree (as described in Section 6.2).

- A commit event ($\langle \textbf{commit } t, T, n \rangle$) indicates that the transaction t has committed. All events that occurred at the transaction path Tt are now visible in its parent T. The logical version n is the commit version of transaction t (as described in Section 6.3.2).
- An advance event ($\langle \textbf{advance } R, T, n \rangle$) indicates that in the resource tree of resource R, at the node corresponding to the transaction path T, the boundary number has been advanced to n (as described in Section 6.2.2).
- A state event ($\langle q_R, T, n \rangle$) indicates that the resource R is in state q at the transaction path T, and the logical version associated with that state is n.

7.3.2 Site Transitions

The site transition relation $\hookrightarrow$ encapsulates the computations of sites. The relation is defined by a judgment, the *site processing judgment*, which takes the following form:

$$\boxed{E \vdash V(\bar{v}) \text{ at } m \text{ in } T \overset{F}{\hookrightarrow} w \text{ at } n}$$

This judgment states that in environment E, when site V is called with arguments $\bar{v}$, and that call has logical version m and is located at transaction path

$$\frac{V(\bar{v}) = w}{E \vdash V(\bar{v}) \text{ at } n \text{ in } T \overset{\emptyset}{\hookrightarrow} w \text{ at } n} \qquad (\textsc{PureSite})$$

Figure 7.12: Site Processing for Pure Computations

T, then the call results in a response w with logical version n, and a set of side effects F.

Each resource that interacts with transactions has its own set of rules which define the site processing judgment for the sites that can access the resource. These rules are given on a per-resource basis in Section 7.7.

Pure sites, as defined in Section 2.3.4, can be handled by a single rule. Since the computations of pure sites are guaranteed to obey atomicity and coatomicity, it is sufficient to map the arguments directly to a response in all cases, ignoring the environment and transactional path entirely, producing no side effects, and reproducing the input logical version in the output. The rule (PureSite), given in Figure 7.12, does exactly that. It is assumed that the mapping (represented by =) is defined elsewhere; this mapping is very simple for the pure sites commonly used in Orc programs, such as conditionals, arithmetic operations, and data structures.

7.3.3 External Transitions

An Orca program f executing within an environment E is denoted by the pair E, f. The *orchestration judgment* defines a transition relation $\longrightarrow$ on such pairs. The orchestration judgment has the following form, where E and E' are environments, and f and f' are Ora expressions:

$$\boxed{E, f \;\longrightarrow\; E', f'}$$

The orchestration judgment is defined by the rules in Figure 7.13.

- The (EnvTau) rule allows a τ transition in the program, producing no change in the environment.
- The (EnvPub) rule similarly allows a publication to emerge from the program, producing no change in the environment.[3]

[3] In an implementation of Ora, a publication might be reported on the console, but here it has no real effect on the environment state.

- The (EnvCall) rule allows the program to make a call to a site, adding a corresponding call event to the environment.
- The (EnvProcess) rule takes a call in the environment and processes it, using the relation $\hookrightarrow$, which defines the behavior of sites. The $\hookrightarrow$ judgment produces three outputs: a response w, a logical version n, and an event set F, the *effects* of the call. The environment is extended with the set of effects, and also with a return event containing w and n. The Ora program is unchanged.

  The (EnvProcess) rule applies only if there is not already a return event for the call in the environment; this enforces the requirement that a site call returns at most once.
- The (EnvRespond) rule detects the completion of a call, and synchronizes with an available response action in the program, so that the response from the site call is substituted for the matching expression $k?$ in the program. The environment is unchanged.

  Since handles are unique, and the $k?w$ removes the expression $k?$ from the program, this rule can be applied at most once for any given handle k.
- The (EnvInit) rule begins a new transaction t in the context of the existing transaction path T (in the case of toplevel transactions, T is ϵ). The program must be ready to create a new transaction, as indicated by its init transition. The maximum init version n among all siblings of this transaction is calculated. The environment is extended with a new init event, located at path T, indicating that T is the parent of the new transaction. It has logical version $n + 1$, which is strictly greater than all of its siblings.
- The (EnvCommit) rule allows a commit in the program to proceed if the transaction t is ready to commit to its parent T in the environment. Most of the logic governing the commitment of transactions is contained in the judgment $\text{commit}(E, Tt) = F$ at n, which will be discussed in Section 7.6. The transition adds a new commit event to the environment, located in the parent T, with logical version n. The presence of this commit event will allow future observers to see the events that took place in the transaction. The commit may also have some side effects F, which are also added to the environment.

- The (EnvAbort) rule allows the transaction t to abort in the program. As long as the transaction t is active in the program, the $\xrightarrow{\textbf{abort}\ Tt}$ transition is available for the program. However, that transition will only occur when this rule applies, and this rule only applies if $\mathrm{conflict}(E, Tt)$ holds, indicating a conflict. The full definition of $\mathrm{conflict}(E, Tt)$ appears in Section 7.6.

  Note that the environment is unchanged by this rule. In an actual implementation, a transaction abort might do a substantial amount of bookkeeping work. In this formal model, there are no cleanup steps; the events associated with the transaction are simply left in the environment, since atomicity guarantees that they will never be observed.

Note that every transition $E, f \longrightarrow E', f'$ of the orchestration judgment obeys the invariant $E \subseteq E'$. In other words, the orchestration judgment is monotonic in the environment: events are never modified or removed, they are only added. In the case of mutable resources, this means that the environment will contain the entire history of states for that resource. In the next section, we will see how this history is summarized to give a well-defined current state to each resource in the environment.

$$\frac{f \xrightarrow{\tau} f'}{E, f \longrightarrow E, f'} \qquad (\textsc{EnvTau})$$

$$\frac{f \xrightarrow{!v \text{ at } n} f'}{E, f \longrightarrow E, f'} \qquad (\textsc{EnvPub})$$

$$\frac{f \xrightarrow{V_k(\bar{v}) \text{ in } T \text{ at } n} f'}{E, f \longrightarrow E \cup \{\langle k \triangleright V(\bar{v}), T, n\rangle\}, f'} \qquad (\textsc{EnvCall})$$

$$\frac{\begin{array}{c} \langle k \triangleright V(\bar{v}), T, m\rangle \in E \\ \langle k \triangleleft \_, \_, \_\rangle \notin E \\ E \vdash V(\bar{v}) \text{ at } m \text{ in } T \overset{F}{\hookrightarrow} w \text{ at } n \end{array}}{E, f \longrightarrow E \cup \{\langle k \triangleleft w, T, n\rangle\} \cup F, f'} \qquad (\textsc{EnvProcess})$$

$$\frac{\begin{array}{c} \langle k \triangleleft w, \_, n\rangle \in E \\ f \xrightarrow{k?w \text{ at } n} f' \end{array}}{E, f \longrightarrow E, f'} \qquad (\textsc{EnvRespond})$$

$$\frac{\begin{array}{c} n = \max(\{i \mid \langle \mathbf{init}\ \_, T, i\rangle \in E\}) \\ f \xrightarrow{\mathbf{init}\ Tt} f' \end{array}}{E, f \longrightarrow E \cup \{\langle \mathbf{init}\ t, T, n+1\rangle\}, f'} \qquad (\textsc{EnvInit})$$

$$\frac{\begin{array}{c} \text{commit}(E, Tt) = F \text{ at } n \\ f \xrightarrow{\mathbf{commit}\ Tt \text{ at } n} f' \end{array}}{E, f \longrightarrow E \cup \{\langle \mathbf{commit}\ t, T, n\rangle\} \cup F, f'} \qquad (\textsc{EnvCommit})$$

$$\frac{\begin{array}{c} \text{conflict}(E, Tt) \\ f \xrightarrow{\mathbf{abort}\ Tt} f' \end{array}}{E, f \longrightarrow E, f'} \qquad (\textsc{EnvAbort})$$

Figure 7.13: Orchestration Judgment

7.4 Environment Operations

The environment defined in the previous section is simply a set of events; it has no structure. Additionally, the environment changes only by adding new events; no existing events are modified or removed. In this section, we define some supporting operations that provide a more structured view of the environment.

7.4.1 Environment Filtering and Mapping

To simplify some of the notation in the semantics, we define some operations for filtering sets of events based on certain criteria, remapping certain characteristics of events, or extracting states from events. These operations are shown in Figure 7.14.

$$\begin{aligned}
E[T] &= \{e \in E \mid e = \langle \_, T, \_ \rangle\} \\
E[< n] &= \{e \in E \mid e = \langle \_, \_, m \rangle \ \wedge \ m < n\} \\
E[R] &= \{e \in E \mid e = \langle q_R, \_, \_ \rangle\} \\
\\
E[\leftarrow n] &= \{\langle c, T, n \rangle \mid \langle c, T, \_ \rangle \in E\} \\
\\
E[*_R] &= \{\!\!\{ q \mid \langle q_R, \_, \_ \rangle \in E \}\!\!\}
\end{aligned}$$

Figure 7.14: Environment Filtering and Remapping Operations

- $E[T]$ selects all events in the set E with the transaction path T. This operation is useful for identifying sets of events that correspond to a ‘node’ in a resource tree.
- $E[< n]$ selects all events in the set E with a logical version strictly less than n.
- $E[R]$ selects all resource state events in E that belong to resource R. Note that $E[R]$ does *not* include **advance** events.
- $E[\leftarrow n]$ is a duplicate of E where each event has the same content and transaction path, but its logical version has been changed to n.
- $E[*_R]$ extracts the multiset[4] of states of resource R in E.

[4] The use of double braces $\{\!\!\{$ ... $\}\!\!\}$ denotes a multiset.

Note that the result of most of these operations is itself a set of events, so the operations can be concatenated: for example, $E[T][< n]$ is the set of all events in E with transaction path T and a logical version strictly less than n.

7.4.2 Image

When a transaction commits, the environment is only changed via the addition of a **commit** event; no other events are moved or copied. The presence of the **commit** event simply indicates that the events of the committed transaction are now visible. The *image judgment* collects the subset of events in the environment that are visible at a particular transaction path, due to the commitment of descendant transactions. It has the following form, where E is an environment and T is a transaction path:

$$\boxed{\text{image}(E, T) = \Sigma}$$

The result Σ is a set of events. However, it connotes a subset, or slice, of an environment, rather than an entire environment, which is why the symbol Σ is used, rather than E.

The image judgment is defined by a single inductive rule, shown in Figure 7.15. The image at a transaction path T is the selection of all events actually located at that path, together with the images of all immediate children that have committed. The image of each child is remapped to have logical version m, where m is the version attached to its commit event.

$$\frac{\Sigma = \bigcup\{\text{image}(E, Tt)[\leftarrow m] \mid \langle \textbf{commit}\ t, T, m\rangle \in E\}}{\text{image}(E, T) = E[T] \cup \Sigma} \qquad (\textsc{Image})$$

Figure 7.15: Image Judgment

7.4.3 State

The external semantics is monotonic in the environment, meaning that it only adds new events; it does not modify or remove old events. As a result, the environment contains the entire history of events, and in particular it contains the entire history of states for a resource. Many operations, such as site computations, need only the current state of a resource. The *state operation* computes the multiset of possible current states $Q_{current}$ of a resource R from a multiset $Q_{history}$ containing the visible history of states for that resource:

$$\boxed{\text{state}_R(Q_{history}) = Q_{current}}$$

The definition of state_R is specific to each resource R; these definitions are given separately for each resource in Section 7.7.

In all legal executions, $Q_{current}$ contains at most one element; the current state of each resource must be unique. This property is the essence of consistency; it will be formalized and discussed further in Section 8.2.

7.5 Observer Semantics

The *observer semantics* is the set of rules that determines the observed state of a resource, as described in Section 6.2.2. It is responsible for enforcing coatomicity by restricting the subset of the environment that is visible to site calls within a transaction.

7.5.1 Snapshot Judgment

The *snapshot judgment* is the core component of the observer semantics. It determines the subset of the environment E that is visible at the transaction path T but not located in the subtree of T; these are the 'outside' events controlled by coatomicity. A snapshot is specific to a resource R. The judgment produces two results: the event subset Σ, and a set of **advance** events G. The judgment is of the following form:

$$\boxed{\text{snapshot}(E, T, R) \overset{G}{\hookrightarrow} \Sigma}$$

The snapshot judgment is defined inductively by the rules shown in Figure 7.16:

- The (Snapshot ϵ) rule states that a snapshot at the root is empty and creates no **advance** events.
- The (Snapshot $\geq$) rule applies to non-root snapshots. It first determines the results of a snapshot for the parent path T. It then identifies the initial version n of the transaction t, and computes the subset of the image at T that is in resource R and has a logical version less than n. The result of the snapshot is the same set of **advance** events as produced by the parent snapshot, and the union of the event subset selected by the parent snapshot with the image subset selected in this rule. The event subset is remapped to transaction path Tt and logical version 0. In other words, the ancestor events in the snapshot should be treated as if they were events in the child, which occurred as early as possible in the child's execution.
- The (Snapshot $<$) rule is different from the (Snapshot $\geq$) rule in only one way: it applies only if the initial version n is less than the boundary number.

$$\mathrm{snapshot}(E, \epsilon, R) \overset{\emptyset}{\hookrightarrow} \emptyset \qquad (\textsc{Snapshot } \epsilon)$$

$$\frac{\begin{array}{c} \mathrm{snapshot}(E,T,R) \overset{G}{\hookrightarrow} \Sigma \\ \langle \mathbf{init}\ t,T,n\rangle \in E \\ \mathrm{image}(E,T)[<n] = \Sigma' \\ \mathrm{boundary}(E,T,R) \geq n \end{array}}{\mathrm{snapshot}(E,Tt,R) \overset{G}{\hookrightarrow} \Sigma \cup \Sigma'} \qquad (\textsc{Snapshot } \geq)$$

$$\frac{\begin{array}{c} \mathrm{snapshot}(E,T,R) \overset{G}{\hookrightarrow} \Sigma \\ \langle \mathbf{init}\ t,T,n\rangle \in E \\ \mathrm{image}(E,T)[<n] = \Sigma' \\ \mathrm{boundary}(E,T,R) < n \end{array}}{\mathrm{snapshot}(E,Tt,R) \overset{G+\langle \mathbf{advance}\ R,T,n\rangle}{\hookrightarrow} \Sigma \cup \Sigma'} \qquad (\textsc{Snapshot } <)$$

Figure 7.16: Snapshot Judgment

The rule behaves identically, except that it adds an **advance** event to the set G.

7.5.2 Boundary Judgment

The snapshot judgment uses another judgment, boundary(E, T, R), which calculates the current boundary number n in the resource tree of R at the node corresponding to transaction path T:

$$\boxed{\mathrm{boundary}(E, T, R) = n}$$

The boundary judgment is defined in Figure 7.17. It simply calculates the maximum logical version among all resource state events in the image of T (this corresponds to increases in the boundary number due to the addition of new states to the node), and all **advance** events located at T (this corresponds to increases in the boundary number due to the state being observed).

$$\frac{\begin{array}{c} I = \{i \mid \langle q_R, \_, i\rangle \in \text{image}(E,T)\} \\ J = \{i \mid \langle \textbf{advance}\ R, T, i\rangle \in E\} \end{array}}{\text{boundary}(E,T,R) = \max(I \cup J)} \quad (\textsc{Boundary})$$

Figure 7.17: Boundary Operation

7.5.3 Observe Judgment

The *observe judgment* computes the observed state of a resource as seen from a particular transaction; it determines the state q of resource R that is observed from transaction path T in environment E. This observation may change some boundary numbers; these changes are represented by a set of **advance** events, G. Lastly, the observed state has a logical version n associated with it. The judgment is of the following form:

$$\boxed{\text{observe}(E,T,R) \overset{G}{\hookrightarrow} q \text{ at } n}$$

The observe judgment is defined in Figure 7.18. It uses two disjoint slices of the environment, Σ and Σ'. The subset Σ is the result of a snapshot at T on resource R. The subset Σ' is the image at T. The observed state q is the unique state of the resource R calculated from the multiset union of the states in Σ and Σ'. If the output of state_R does not contain exactly one element, then the judgment is undefined. The logical version n is the maximum logical version in Σ'; logical versions of events from Σ are ignored.

$$\frac{\begin{array}{c} \text{snapshot}(E,T,R) \overset{G}{\hookrightarrow} \Sigma \\ \text{image}(E,T) = \Sigma' \\ \text{state}_R(\Sigma[*_R] \uplus \Sigma'[*_R]) = \{\!\{q\}\!\} \\ n = \max(\ \{i \mid \langle q_R, \_, i\rangle \in \Sigma'\}\ ) \end{array}}{\text{observe}(E,T,R) \overset{G}{\hookrightarrow} q \text{ at } n} \quad (\textsc{Observe})$$

Figure 7.18: Observe Judgment

7.6 Commit Semantics

The *commit semantics* is the set of rules that determines whether and when a transaction may commit or abort. The commit and conflict judgments used in the (EnvCommit) and (EnvAbort) rules of the external semantics are defined here.

7.6.1 Merge Operation

When a transaction commits to its parent, the state for a resource in the child may have diverged from the state for that resource in the parent. The *merge operation* takes these divergent states and determines whether they can be reconciled. It is written as follows:

$$\boxed{\mathrm{merge}_R(q_{branch}, q_{parent}, q_{child}) = \textbf{pass} \mid q \mid \textbf{fail}}$$

The state q_{branch} is the state of R in the parent at the point when the child transaction began to modify R. The state q_{parent} is the current state in the parent, and the state q_{child} is the current state in the child. The operation has three possible results:

1. A result of **pass** indicates that the states may coexist in the subsequent history of states for R, without modification.
2. A result of q indicates that the states may coexist in the subsequent history of states for R, but only if the state q is added.
3. A result of **fail** indicates that the states are incompatible; there is no valid history for R that can contain both states.

As with state_R, the definition of merge_R is specific to each resource R. The definitions are given separately for each resource in Section 7.7.

All definitions of merge_R have the following property:

$$\mathrm{merge}_R(q, q, q') = \mathrm{merge}_R(q, q', q) = \textbf{pass}$$

That is, if the state has changed only in the parent or in the child since the branch, the merge always succeeds with no changes needed to the environment.

7.6.2 Graft Judgment

The *graft judgment* is the core judgment of the commit semantics:

$$\boxed{\text{graft}(E, Tt, R) = \textbf{pass} \mid q \mid \textbf{fail}}$$

This judgment states that in environment E, if the events at resource R in the image of transaction path Tt had occurred at transaction path T instead, then they would form a consistent execution if the result is **pass**, or they can be reconciled to a consistent execution with the addition of q if the result is q, or they would fail to form a consistent execution if the result is **fail**.

The graft judgment is defined by a single rule, shown in Figure 7.19. This rule computes three slices of the environment: the branching point (Σ_{branch}), the parent (Σ_{parent}), and the child (Σ_{child}). It the computes the corresponding state for each of these slices, and passes those states to the merge_R operation, which produces the result.

$$\frac{\begin{array}{c}\text{snapshot}(E, Tt, R) \overset{G}{\hookrightarrow} \Sigma_{branch}\\ \Sigma_{parent} = \Sigma_{branch} \cup \text{image}(E, T)\\ \Sigma_{child} = \Sigma_{branch} \cup \text{image}(E, Tt)\\ \text{state}_R(\Sigma_{branch}[*_R]) = \{\!\{q_{branch}\}\!\}\\ \text{state}_R(\Sigma_{parent}[*_R]) = \{\!\{q_{parent}\}\!\}\\ \text{state}_R(\Sigma_{child}[*_R]) = \{\!\{q_{child}\}\!\}\end{array}}{\text{graft}(E, Tt, R) = \text{merge}_R(q_{branch}, q_{parent}, q_{child})} \quad (\textsc{Graft})$$

Figure 7.19: Graft Judgment

7.6.3 Participants

The set of *participants* in a commit operation is the set of resources that could have a state in the child transaction that diverges from the state in the parent transaction:

$$\text{participants}(E, Tt) = \{R \mid \text{image}(E, Tt)[*_R] \neq \emptyset\}$$

The set of participants includes each resource R that has a modified state in the image of Tt. If R is not in the set of participants, we know that no divergence is possible[5] since the state has either changed only in T, or not at all.

[5] For all non-participants, $q_{branch} = q_{child}$. It follows that $\text{merge}_R(q_{branch}, q_{parent}, q_{child}) = \textbf{pass}$.

$$\frac{\begin{array}{c} I = \{\text{boundary}(E, T, R) \mid R \in \text{participants}(E, Tt)\} \\ \langle \textbf{init}\ t, T, n \rangle \in E \end{array}}{\text{boundary}_\star(E, Tt) = \max(I \cup \{n\})} \quad (\textsc{Boundary}\star)$$

Figure 7.20: Boundary⋆ Judgment

7.6.4 Boundary⋆ Judgment

The $\text{boundary}_\star$ judgment determines the commit version of a transaction, as described in Section 6.3. It is written as follows:

$$\boxed{\text{boundary}_\star(E, Tt) = n}$$

The $\text{boundary}_\star$ judgment states that in environment E, if Tt commits, it must be assigned the logical version n as its commit number. The judgment is defined by the rule shown in Figure 7.20. It calculates n as the maximum boundary number of all participants; that is, all resources in T that will be modified by events in Tt, if Tt commits. Atomicity requires that every event from Tt have an identical logical version in T. Coatomicity requires that an event modifying a resource be given a logical version no less than that resource's boundary number. Furthermore, the smallest possible logical version should be selected. Given these constraints, n is the only choice.

7.6.5 Commit Judgment

With these supporting judgments, we can now define the *commit judgment*, which is used in the (EnvCommit) rule of the external semantics.

$$\boxed{\text{commit}(E, Tt) = F \text{ at } n}$$

The commit judgment states that in environment E, if all of the events in the image of Tt had instead occurred at transaction path T with logical version n, and the events F were also added, then the resulting execution would still be consistent. The commit judgment is defined by the rule shown in Figure 7.21. For each resource in the set of participants, grafting that resource must not result in a **fail**. F is the set of events corresponding to the state changes needed by each graft, each with transaction path T and logical version n.

$$\frac{\begin{array}{c}\text{participants}(E, Tt) = \mathcal{R} \\ \text{boundary}_\star(E, Tt) = n \\ \forall R : R \in \mathcal{R} : \text{graft}(E, Tt, R) \neq \textbf{fail} \\ F = \{\langle q_R, T, n\rangle \mid R \in \mathcal{R} \ \wedge \ \text{graft}(E, Tt, R) = q\}\end{array}}{\text{commit}(E, Tt) = F \text{ at } n} \qquad (\textsc{Commit})$$

Figure 7.21: Commit Judgment

$$\frac{\begin{array}{c}R \in \text{participants}(E, Tt) \\ \text{graft}(E, Tt, R) = \textbf{fail}\end{array}}{\text{conflict}(E, Tt)} \qquad (\textsc{Conflict})$$

Figure 7.22: Conflict Judgment

7.6.6 Conflict Judgment

The *conflict judgment* detects when it is impossible for a transaction to commit. It is used in the (EnvAbort) rule of the external semantics, to enable a transaction abort.

$$\boxed{\text{conflict}(E, Tt)}$$

The conflict judgment states that the commit judgment does not hold for transaction Tt in environment E. The rule (Conflict), shown in Figure 7.22, defines the conflict judgment. It simply requires that **fail** occur in the graft judgment of some resource in the set of participants.

Conflict is a stable property: if conflict(E, Tt) holds, then for all E' such that $E \subseteq E'$, conflict(E', Tt) also holds.

7.6.7 The `Abort` site

The `Abort` site, introduced in Section 5.1.2, requires two special rules in order to function correctly. These rules are given in Figure 7.23. If `Abort` is called from a non-root transaction, it causes its host transaction to abort; this is handled by (AbortSite), an external semantics rule. If `Abort` is called from the root, it simply halts; this is handled by (AbortSiteRoot), a site processing rule.

$$\frac{\begin{array}{c}\langle\, \_ \triangleright \texttt{Abort}(), Tt, \_\,\rangle \in E \\ f \ \overset{\textbf{abort}}{\longrightarrow}^{Tt} \ f'\end{array}}{E, f \ \longrightarrow \ E, f'} \tag{\textsc{AbortSite}}$$

$$E \vdash \texttt{Abort}() \text{ at } n \text{ in } \epsilon \ \overset{\emptyset}{\hookrightarrow} \ \texttt{stop} \text{ at } n \tag{\textsc{AbortSiteRoot}}$$

Figure 7.23: Rules for the `Abort` Site

7.7 Resource Semantics

This section defines, for each resource type, the behavior of its resource-specific and site-specific operations (state_R, merge_R, and $\hookrightarrow$), as well as the grammar of its resource states.

7.7.1 Guard

States

$$\begin{array}{rcl} q & \in & \textit{Resource State} \quad ::= \quad ... \mid \textsc{Guard}(b) \\ b & \in & \textit{Boolean} \end{array}$$

State Operation

Let $\beta = \bigvee\{b \mid \textsc{Guard}(b) \in Q\}$.

If β is false, then $\mathrm{state}_R(Q) = \{\!\{\textsc{Guard}(\mathbf{false})\}\!\}$.

If β is true, then $\mathrm{state}_R(Q) = \{\!\{\textsc{Guard}(\mathbf{true}) \in Q\}\!\}$.

Merge Operation

If

$$\begin{aligned} q_s &= \textsc{Guard}(b_s) \\ q_p &= \textsc{Guard}(b_p) \\ q_c &= \textsc{Guard}(b_c) \end{aligned}$$

Then

$$\mathrm{merge}_R(q_s, q_p, q_c) = \mathbf{pass} \qquad \text{if } b_p \wedge b_c \implies b_s$$

$$\mathrm{merge}_R(q_s, q_p, q_c) = \mathbf{fail} \qquad \text{otherwise}$$

Site Processing Rules

$$\frac{\begin{array}{c} R \text{ fresh in } E \\ e = \langle \textsc{Guard}(\mathbf{false})_R, T, n \rangle \end{array}}{E \vdash \texttt{Guard}() \text{ at } n \text{ in } T \ \overset{e}{\hookrightarrow} \ \mathit{Guard}_R \text{ at } n} \qquad (\textsc{GuardSite})$$

$$\frac{\begin{array}{c} \text{observe}(E, T, R) \ \overset{G}{\hookrightarrow} \ \textsc{Guard}(\mathbf{false}) \text{ at } m' \\ n = \max(m, \text{boundary}(E, T, R)) \\ e = \langle \textsc{Guard}(\mathbf{true})_R, T, n \rangle \end{array}}{E \vdash \mathit{Guard}_R() \text{ at } m \text{ in } T \ \overset{G+e}{\hookrightarrow} \ \texttt{signal} \text{ at } n} \qquad (\textsc{GuardSiteV})$$

$$\frac{\begin{array}{c} \text{observe}(E, T, R) \ \overset{G}{\hookrightarrow} \ \textsc{Guard}(\mathbf{true}) \text{ at } m' \\ n = \max(m, m') \end{array}}{E \vdash \mathit{Guard}_R() \text{ at } m \text{ in } T \ \overset{G}{\hookrightarrow} \ \texttt{stop} \text{ at } n} \qquad (\textsc{GuardSiteN})$$

Figure 7.24: Site Processing Rules for `Guard`

7.7.2 Ref

States

$$\begin{array}{rcl} q & \in & \textit{Resource State} \quad ::= \quad ... \mid \textsc{Ref}(\boxed{v}, \rho) \\ \boxed{v} & \in & \square \mid v \\ \rho & \in & \mathbb{N} \end{array}$$

State Operation

$$\begin{aligned} \text{state}_R(Q) = \{\!\{ \ & \textsc{Ref}(\boxed{v}, \rho) \in Q \\ & \mid \textsc{Ref}(\_, \rho') \in Q \implies \rho' \leq \rho \ \}\!\} \end{aligned}$$

Merge Operation

If

$$\begin{aligned} q_s &= \textsc{Ref}(\_, \rho_s) \\ q_p &= \textsc{Ref}(\_, \rho_p) \\ q_c &= \textsc{Ref}(\_, \rho_c) \end{aligned}$$

then

$$\begin{array}{ll} \text{merge}_R(q_s, q_p, q_c) = \textbf{pass} & \text{if } \rho_p = \rho_s \\ \text{merge}_R(q_s, q_p, q_c) = \textbf{pass} & \text{if } \rho_c = \rho_s \\ \text{merge}_R(q_s, q_p, q_c) = \textbf{fail} & \text{otherwise} \end{array}$$

Site Processing Rules

$$\frac{\begin{array}{c} R \text{ fresh in } E \\ e = \langle \textsc{Ref}(\Box, 0)_R, T, n\rangle \\ v = \{.\ \ \texttt{read = } \mathit{Read}_R\texttt{, write = } \mathit{Write}_R\ \texttt{.}\} \end{array}}{E \vdash \texttt{Ref}() \text{ at } n \text{ in } T \overset{\{e\}}{\hookrightarrow} v \text{ at } n} \qquad (\textsc{RefNSite})$$

$$\frac{\begin{array}{c} R \text{ fresh in } E \\ e = \langle \textsc{Ref}(u, 0)_R, T, n\rangle \\ v = \{.\ \ \texttt{read = } \mathit{Read}_R\texttt{, write = } \mathit{Write}_R\ \texttt{.}\} \end{array}}{E \vdash \texttt{Ref}(u) \text{ at } n \text{ in } T \overset{\{e\}}{\hookrightarrow} v \text{ at } n} \qquad (\textsc{RefVSite})$$

$$\frac{\begin{array}{c} \text{observe}(E, T, R) \overset{G}{\hookrightarrow} \textsc{Ref}(v, \_) \text{ at } m' \\ n = \max(m, m') \end{array}}{E \vdash \mathit{Read}_R() \text{ at } m \text{ in } T \overset{G}{\hookrightarrow} v \text{ at } n} \qquad (\textsc{ReadSite})$$

$$\frac{\begin{array}{c} \text{observe}(E, T, R) \overset{G}{\hookrightarrow} \textsc{Ref}(\_, \rho) \text{ at } m' \\ n = \max(m, \text{boundary}(E, T, R)) \\ e = \langle \textsc{Ref}(v, \rho + 1)_R, T, n\rangle \end{array}}{E \vdash \mathit{Write}_R(v) \text{ at } m \text{ in } T \overset{G+e}{\hookrightarrow} \texttt{signal} \text{ at } n} \qquad (\textsc{WriteSite})$$

Figure 7.25: Site Processing Rules for `Ref`

7.7.3 Cell

The `Cell` site uses the same resource states and the same state and merge operations as the `Ref` site. Only the site processing rules differ.

Site Processing Rules

$$\frac{\begin{array}{c} R \text{ fresh in } E \\ e = \langle \textsc{Ref}(\Box, 0)_R, T, n \rangle \\ v = \texttt{\{.}\ \ \texttt{read = } \mathit{Read}_R\texttt{,}\ \ \texttt{write = } \mathit{Cellset}_R\ \ \texttt{.\}} \end{array}}{E \vdash \texttt{Cell}() \text{ at } n \text{ in } T \overset{\{e\}}{\hookrightarrow} v \text{ at } n} \tag{\textsc{CellSite}}$$

$$\frac{\begin{array}{c} \text{observe}(E, T, R) \overset{G}{\hookrightarrow} \textsc{Ref}(\Box, 0) \text{ at } m' \\ n = \max(m, \text{boundary}_R(E, T)) \\ e = \langle \textsc{Ref}(v, 1)_R, T, n \rangle \end{array}}{E \vdash \mathit{Cellset}_R(v) \text{ at } m \text{ in } T \overset{G+e}{\hookrightarrow} \texttt{signal} \text{ at } n} \tag{\textsc{CellsetNSite}}$$

$$\frac{\begin{array}{c} \text{observe}(E, T, R) \overset{G}{\hookrightarrow} \textsc{Ref}(v, \_) \text{ at } m' \\ n = \max(m, m') \end{array}}{E \vdash \mathit{Cellset}_R(\_) \text{ at } m \text{ in } T \overset{G}{\hookrightarrow} \texttt{stop} \text{ at } n} \tag{\textsc{CellsetVSite}}$$

Figure 7.26: Site Processing Rules for `Cell`

7.7.4 Channel

State Grammar

$q \in$ *Resource State* $::=$ ... **|** $\textsc{Channel}(\bar{v}, \rho, \rho)$
$\rho \in \mathbb{N}$

State Operation

$$\begin{aligned}\text{state}_R(Q) = \{\!\!\{\ &\textsc{Channel}(\bar{v}, g, p) \in Q \\ &\mid \textsc{Channel}(\_, g', p') \in Q \implies g' + p' \leq g + p\}\!\!\}\end{aligned}$$

Merge Operation

If

$$\begin{aligned}q_s &= \textsc{Channel}(\_, g_s, p_s) \\ q_a &= \textsc{Channel}(u_0 \ldots u_i, g_a, p_a) \\ q_b &= \textsc{Channel}(v_0 \ldots v_j, g_b, p_b)\end{aligned}$$

then

$$
\begin{aligned}
\mathrm{merge}_R(q_s, q_a, q_b) &= \textbf{pass} && \text{if } g_a = g_s \wedge p_a = p_s \\
&= \textbf{pass} && \text{if } g_b = g_s \wedge p_b = p_s \\
&= \textsc{Channel}(u_k \ldots u_i, g_b, p_a) && \text{if } g_a = g_s \wedge p_b = p_s \\
&\qquad \text{where } k = g_b - g_s \\
&= \textsc{Channel}(v_k \ldots v_j, g_a, p_b) && \text{if } g_b = g_s \wedge p_a = p_s \\
&\qquad \text{where } k = g_a - g_s \\
&= \textbf{fail} && \text{otherwise}
\end{aligned}
$$

Site Processing Rules

$$\frac{\begin{array}{c} R \text{ fresh in } E \\ e = \langle \textsc{Channel}(\epsilon, 0, 0)_R, T, n \rangle \\ v = \texttt{\{.}\ \ \texttt{get = } \mathit{Get}_R\texttt{, put = } \mathit{Put}_R\ \ \texttt{.\}} \end{array}}{E \vdash \texttt{Channel}() \text{ at } n \text{ in } T \overset{\{e\}}{\hookrightarrow} v \text{ at } n} \qquad (\textsc{ChannelSite})$$

$$\frac{\begin{array}{c} \text{observe}(E, T, R) \overset{G}{\hookrightarrow} \textsc{Channel}(u\bar{v}, g, p) \text{ at } m' \\ n = \max(m, \text{boundary}(E, T, R)) \\ e = \langle \textsc{Channel}(\bar{v}, g+1, p)_R, T, n \rangle \end{array}}{E \vdash \mathit{Get}_R() \text{ at } m \text{ in } T \overset{G+e}{\hookrightarrow} u \text{ at } n} \qquad (\textsc{GetSite})$$

$$\frac{\begin{array}{c} \text{observe}(E, T, R) \overset{G}{\hookrightarrow} \textsc{Channel}(\bar{v}, g, p) \text{ at } m' \\ n = \max(m, \text{boundary}(E, T, R)) \\ e = \langle \textsc{Channel}(\bar{v}u, g, p+1)_R, T, n \rangle \end{array}}{E \vdash \mathit{Put}_R(u) \text{ at } m \text{ in } T \overset{G+e}{\hookrightarrow} \texttt{signal} \text{ at } n} \qquad (\textsc{PutSite})$$

Figure 7.27: Site Processing Rules for `Channel`

7.7.5 Semaphore

States

$q \in$ *Resource State* $::=$ ... | $\textsc{Semaphore}(\rho, \rho)$
$\rho \in \mathbb{N}$

State Operation

$$\begin{aligned}\text{state}_R(Q) = \{\!\{ &\ \textsc{Semaphore}(a, r) \in Q \\ &\mid \textsc{Semaphore}(a', r') \in Q \implies a' + r' \leq a + r \}\!\}\end{aligned}$$

Merge Operation

If

$$\begin{aligned}q_s &= \textsc{Semaphore}(a_s, r_s) \\ q_p &= \textsc{Semaphore}(a_p, r_p) \\ q_c &= \textsc{Semaphore}(a_c, r_c)\end{aligned}$$

then

$$\text{merge}_R(q_s, q_p, q_c) = \textbf{pass} \qquad \text{if } a_p = a_s \wedge r_p = r_s$$

$$\text{merge}_R(q_s, q_p, q_c) = \textbf{pass} \qquad \text{if } a_c = a_s \wedge r_c = r_s$$

$$\begin{aligned}\text{merge}_R(q_s, q_p, q_c) = &\ \textsc{Semaphore}(a_m, r_m) \qquad \text{if } a_m \leq r_m \\ &\quad \text{where } a_m = a_p + a_c - a_s \\ &\quad \text{and } r_m = r_p + r_c - r_s\end{aligned}$$

$$\text{merge}_R(q_s, q_p, q_c) = \textbf{fail} \qquad \text{otherwise}$$

Site Processing Rules

$$\frac{\begin{array}{c} R \text{ fresh in } E \\ e = \langle \text{\textsc{Semaphore}}(0,i)_R, T, n\rangle \\ v = \texttt{\{.\ \ acquire = }\mathit{Acquire}_R\texttt{, release = }\mathit{Release}_R\texttt{ .\}} \end{array}}{E \vdash \texttt{Semaphore}(i) \text{ at } n \text{ in } T \overset{\{e\}}{\hookrightarrow} v \text{ at } n} \qquad (\text{\textsc{SemaphoreSite}})$$

$$\frac{\begin{array}{c} \text{observe}(E,T,R) \overset{G}{\hookrightarrow} \text{\textsc{Semaphore}}(a,r) \text{ at } m' \\ a < r \\ n = \max(m, \text{boundary}(E,T,R)) \\ e = \langle \text{\textsc{Semaphore}}(a+1,r)_R, T, n\rangle \end{array}}{E \vdash \mathit{Acquire}_R() \text{ at } m \text{ in } T \overset{G+e}{\hookrightarrow} \texttt{signal} \text{ at } n} \qquad (\text{\textsc{AcquireSite}})$$

$$\frac{\begin{array}{c} \text{observe}(E,T,R) \overset{G}{\hookrightarrow} \text{\textsc{Semaphore}}(a,r) \text{ at } m' \\ n = \max(m, \text{boundary}(E,T,R)) \\ e = \langle \text{\textsc{Semaphore}}(a,r+1)_R, T, n\rangle \end{array}}{E \vdash \mathit{Release}_R() \text{ at } m \text{ in } T \overset{G+e}{\hookrightarrow} \texttt{signal} \text{ at } n} \qquad (\text{\textsc{ReleaseSite}})$$

Figure 7.28: Site Processing Rules for `Semaphore`

Chapter 8

Formal Properties of Ora

The dual properties of atomicity and coatomicity were introduced informally in Chapter 4, and the property of consistency was introduced informally in Chapter 5. In this chapter, we formalize each of those properties, and prove that Ora, as defined by the formal semantics in Chapter 7, has those properties.

8.1 Atomicity and Coatomicity

The Ora language is designed to maintain the properties of atomicity and coatomicity for each use of the **atomic** combinator. These properties were described informally in Chapter 4; in this section we will define them formally. To do so, we will need to define four supporting relations: *internal causality*, *external causality*, *virtual causality*, and *relevance*. With these relations defined, we can then say that Ora guarantees atomicity and coatomicity if a particular relationship holds for relevance, virtual causality, and external causality.

8.1.1 Notation

Before defining these relations, we will need to define some supporting notation.

Histories

Some of the relations are defined in the context of a particular execution. We will use the variable name H (for 'history') as a shorthand for a whole execution of the form $\emptyset, f \ \longrightarrow^* \ E, g$. And given an execution H, we will write $\mathcal{E}(H)$ to refer to the final environment E in that execution.

Event Components

Recall from Section 7.3.1 that events in Ora are triples of the form $\langle c, T, n\rangle$, where c is the content of the event, T is its transaction path, and n is its logical version. We define the following operations to extract each of these individual components of an event:

$$\begin{aligned}
\mathcal{C}(\langle c, \_, \_\rangle) &= c \\
\mathcal{P}(\langle \_, T, \_\rangle) &= T \\
\mathcal{V}(\langle \_, \_, n\rangle) &= n
\end{aligned}$$

We also define another operation to extract the handle k associated with a site call or site return event. It is undefined for other event types.

$$\begin{aligned}
\mathcal{K}(\langle k \triangleright \_, \_, \_\rangle) &= k \\
\mathcal{K}(\langle k \triangleleft \_, \_, \_\rangle) &= k
\end{aligned}$$

8.1.2 Internal Causality

Internal causality is a strict partial order on site call handles that represents the flow of control through the execution of an Ora program. It is written as follows, where k and k' are call handles and H is the execution in which they occur:

$$\boxed{H \vdash k < k'}$$

Since the internal semantics of Ora do not create events in the environment corresponding to the individual evaluation steps of the program, we must represent the causality of Ora indirectly, hence the choice of an ordering on site call handles. If $k < k'$, this means that the return of the site call labeled by k triggered a chain of evaluation steps within the program that resulted in the initiation of the site call labeled by k'.

8.1.3 External Causality

External causality is the relation between events that represents the flow of information among steps of the external semantics and state changes in the environment. It is written as follows, where x and y are events and H is the execution in which they occur:

$$\boxed{H \vdash x \prec y}$$

External causality is a formalization of the solid edges in the event graphs of Chapter 4.

Definition

External causality is defined by the following rules:

- Whenever a site call event y is added to the environment, then for every site return event x such that $\mathcal{K}(x) < \mathcal{K}(y)$, let $x \prec y$.

- Whenever a site call adds side effects F to the environment, identify the site call event x that began the call, and for each y in F, let $x \prec y$.

- Whenever a site call adds side effects F to the environment, if the observe judgment was used in the site processing step, identify the state q that was observed, identify the event $x = \langle q_R, \_, \_\rangle$ from which it was taken, and for each y in F, let $x \prec y$.

- Whenever a site return event y is added to the environment, find the unique call event x such that $\mathcal{K}(x) = \mathcal{K}(y)$, and let $x \prec y$.

- Whenever a site return event y is added to the environment, if the observe judgment was used in the site processing step, identify the state q that was observed, identify the event $x = \langle q_R, \_, \_\rangle$ from which it was taken, and let $x \prec y$.

- Whenever a site return event y is added to the environment, for each event x in the set of side effects F also added to the environment by the same (EnvProcess) step, let $x \prec y$.

- Whenever a transaction is committed, for each y in the set of merge events F added to the environment, identify the states q and q' that were merged to produce the event y, find the events x and x' corresponding to those states, and let $x \prec y$ and $x' \prec y$.

Note that **init**, **commit**, and **advance** events are never included in the $\prec$ relation. These are considered to be 'invisible' events, since they would not occur in a nontransactional execution; they are simply present to aid in the enforcement of atomicity and coatomicity, and do not affect the actual meaning of the program.

Transitive Closure

External causality is not an ordering relation, since it is only irreflexive and asymmetric; it is not transitive. Its transitive closure, however, is a strict partial order. The transitive closure of $\prec$ is written as follows:

$$\boxed{H \vdash x \prec^* y}$$

Logical Versions

Observe that whenever a new event is added to the environment, its logical version is computed by the semantics to be at least as large as the logical version of each of its causes. We can capture this observation in a simple property:

Conjecture 1 (Logical Version Conjecture)**.**

$$H \vdash x \prec y \implies \mathcal{V}(x) \leq \mathcal{V}(y)$$

Note that $H \vdash x \prec^* y \implies \mathcal{V}(x) \leq \mathcal{V}(y)$ follows immediately as a corollary.

8.1.4 Virtual Causality

Virtual causality is the relation between events that represents the requirements imposed by atomicity and coatomicity. It is written as follows, where x is the *virtual cause*, y is the *virtual effect*, and H is the execution in which x and y occur:

$$\boxed{H \vdash x \mathbin{\bigcirc\mkern-15mu\prec} y}$$

Virtual causality is a formalization of the dotted edges in the event graphs of Chapter 4. The relation is defined inductively by the rules in Figure 8.1:

- (Root) ensures that the $\text{\textcircled{$\prec$}}$ relation is a superset of the $\prec^*$ relation.
- (Atomic) allows an event to share its virtual effects with any other event in the same committed subtree of the transaction tree, hence the use of the image judgment. The logical version of x may have been remapped in the image, hence we allow an arbitrary version n. The virtual effect must be outside of the subtree.
- (Coatomic) allows an event to share its virtual causes with any other event in the same subtree of the transaction tree. The virtual cause must be outside of the subtree.
- (Trans) makes the $\text{\textcircled{$\prec$}}$ relation transitive.

$$\frac{H \vdash x \prec^* y}{H \vdash x \text{\textcircled{$\prec$}} y} \tag{\textsc{Root}}$$

$$\frac{H \vdash z \text{\textcircled{$\prec$}} y \qquad \mathcal{P}(z) \not\leq \mathcal{P}(y) \qquad x[\leftarrow n] \in \mathrm{image}(\mathcal{E}(H), \mathcal{P}(z))}{H \vdash x \text{\textcircled{$\prec$}} y} \tag{\textsc{Atomic}}$$

$$\frac{H \vdash y \text{\textcircled{$\prec$}} z \qquad \mathcal{P}(z) \not\leq \mathcal{P}(y) \qquad \mathcal{P}(z) \leq \mathcal{P}(x)}{H \vdash y \text{\textcircled{$\prec$}} x} \tag{\textsc{Coatomic}}$$

$$\frac{H \vdash x \text{\textcircled{$\prec$}} z \qquad H \vdash z \text{\textcircled{$\prec$}} y}{H \vdash x \text{\textcircled{$\prec$}} y} \tag{\textsc{Trans}}$$

Figure 8.1: Virtual Causality

While (Atomic) and (Coatomic) do not appear to be duals, their underlying requirements are more symmetric than they seem. Since x is required to be in the image at $\mathcal{P}(z)$, it follows that $\mathcal{P}(z) \leq \mathcal{P}(x)$, which is the same premise as for (Coatomic). While x can be present in the image only if there is a sequence of **commit** events from $\mathcal{P}(x)$ to $\mathcal{P}(z)$, we can think of that sequence as the dual of the sequence of **init** events leading from $\mathcal{P}(z)$ to $\mathcal{P}(x)$ in the (Coatomic) case.

8.1.5 Relevance

We need one more relation on events, the *relevance* relation, written as follows:

$$\boxed{x \prec\!\!\prec y}$$

The relevance relation expresses the possibility that an event with the same content as x could cause an event with the same content as y in some execution, even if $x \prec y$ does not hold in the execution where x and y occurred. For example, a write event on a reference is always relevant to a read event on that same reference, even if the write was not seen by the read, and regardless of the transaction path or logical version of each event.

Formally, we define the relevance relation in the following way:

If there exists some program f, some execution $\emptyset, f \longrightarrow^* E, g$ of that program, and some events x', y' such that

$$x' \in E$$
$$y' \in E$$
$$x' \prec y'$$

then for all events x, y such that

$$\mathcal{C}(x) = \mathcal{C}(x')$$
$$\mathcal{C}(y) = \mathcal{C}(y')$$

we say that $x \prec\!\!\prec y$.

Note that $\prec\!\!\prec$ is defined in terms of $\prec$, not $\prec^*$, since transitivity may relate all kinds of events, but those relationships are not necessarily relevant; we want only the root causalities, not the derived ones. Also note that $\prec\!\!\prec$, unlike $\prec$ and $\textcircled{$\prec$}$, is not defined in the context of a particular execution, since its definition quantifies over all possible executions.

8.1.6 The Bubble Conjecture

Now that we have defined external causality, virtual causality, and relevance, we can formally define atomicity and coatomicity.

We say that Ora guarantees atomicity and coatomicity if the following property holds for all Ora programs f:

Conjecture 2 (Bubble Conjecture)**.** *Suppose* $H = \emptyset, f \quad \longrightarrow^* \quad E, g$. *Then for all events* x *and* y *in* E*,*

$$x \lll y \implies (H \vdash x \bigcirc\!\!\!\!\!\prec\; y \iff H \vdash x \prec^* y)$$

This means that at every step of every possible execution of f, if we erase all irrelevant causalities, then virtual causality and external causality are identical. This is exactly the property described in Chapter 4.

8.2 Consistency

In this section, we will formally define the property of *consistency*, which was described only informally in the preceding chapters.

The need for an additional formal property arises from the insufficiency of atomicity and coatomicity to restrict certain erroneous behaviors in Ora programs. While atomicity and coatomicity do represent the indivisibility and shared causality of event sets, the introduction of the **atomic** combinator to allow such evaluations also augments the underlying Orc language with additional expressive power: the state of a resource is allowed to diverge in different transactions. If a transaction commits after such a divergence, it is possible for a subsequent site call to observe the divergence, and thus be unable to determine a unique state for the resource.

We have already seen an example of this problem in Section 4.2.2. Consider the example program from that section:

```
val r = Ref(0)
r? << atomic ( atomic (r := r? + 1) & atomic (r := r? + 1) )
```

A possible event graph of its execution is shown in Figure 8.2. This graph conforms to both atomicity and coatomicity. However, the state observed by the outermost `r?` operation is divergent; both writes are seen, but rather than observing the value 2, two instances of the value 1 are seen. This means that the result of state$_R$ for that resource contains two elements, not one, and so the state of the resource is not well defined. Choosing one of the states arbitrarily is not permitted, and even if it were, both choices are incorrect.

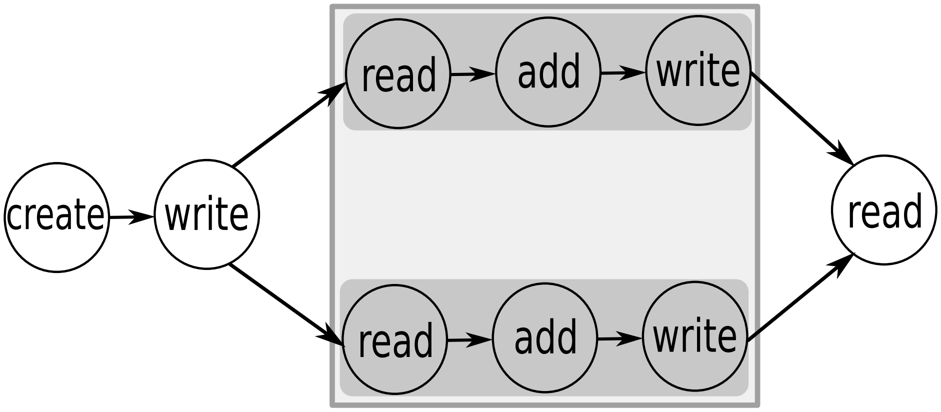


Figure 8.2: Divergent writes (inconsistent)

To formally express consistency, we would like to impose the requirement that state_R produces a unique result. However, this is not an unconditional requirement; the multiset of states given as an argument must be an observable set of states according to atomicity and coatomicity. If a transaction has not yet committed, and thus its events are not observable by outside events (due to atomicity), then its states may diverge. In many cases (such as when using channels), this divergence of states is not an error; it can be resolved during the commit by the merge_R operation, which introduces a new state that reconciles the divergent states of parent and child. This is the essential function of merge_R; it ensures that state_R can report a single state that is correctly representative of the whole history of states.

Furthermore, the consistency property should be conditional on the availability of the resource R; we must dismiss cases where the result of state_R would be $\emptyset$ simply because the resource does not exist yet.

Fortunately, the observe judgment (defined in Section 7.5.3) is already structured in such a way that it enforces most of the needed requirements; the judgment assembles the multiset of states for an observer, and it is only defined if that multiset has exactly one element. With a simple condition to account for the availability of the resource, we can express consistency formally in terms of the observe judgment.

Let M_R denote any site that operates on the resource R. Ora is said to be *consistent* if the following conjecture holds for all Ora programs f:

Conjecture 3 (Consistency Conjecture)**.** *Suppose that $\emptyset, f \longrightarrow^* E, g$. Then it follows that*

$$\langle k \triangleright M_R(\_), T, \_\rangle \in E \implies \text{observe}(E, T, R) \overset{G}{\hookrightarrow} q \text{ at } n$$

for some G, q, n.

A call to a site M_R can only occur if a value containing M_R has previously been created and published. Since this only occurs in resource creation rules (which also initialize the resources they create) we know that the presence of M_R indicates that the resource R has some valid initial state.

Chapter 9

Discussion

In this final chapter, we will consider a few of Ora's design decisions and limitations in greater depth. For example, we will discuss why Ora has the particular set of transactional resources that it does. The semantics of Ora does not currently incorporate time, and `Rwait` is not included in the theory; we will explore how time might be added. In Ora, operations may block within a transaction; we will talk about why this feature was included, and consider the problems that it sometimes causes.

9.1 Choosing Resources

The Ora language supports five sites that create transactional resources: `Guard`, `Ref`, `Cell`, `Semaphore`, and `Channel`. A question naturally arises: why were these specific resources chosen?

The creation of mutable references with `Ref` is a natural starting point, since this allows Ora to match the expressive power of existing transactional memory systems. The inclusion of `Guard` was necessary to define atomic choice. The addition of `Cell` (and uses of `Ref` with no initial value) demonstrates that Ora allows transactions to have blocking operations; the implications of this inclusion are discussed further in Section 9.3. Including `Semaphore` and `Channel` continues this trend, and also demonstrates another key capability of Ora: the use of resources with multiple mutative operations. For example, a channel has `put` and `get` operations, which both change the state of the channel, but do not necessarily conflict with each other; a child transaction could perform `get` operations while its parent performs `put` operations, but a successful commit is still possible, using a merge operation.

These particular resources were also chosen because they are the most frequently used stateful primitives in the Orc standard library [Orc13c]. As such, they provide Ora with useful benchmarks for its ability to support the various coding idioms of Orc. Furthermore, having a diverse set of resources ensures that the site semantics of Ora are not overly specialized.

Of course, there are many other use cases that might not be covered directly by these resources. Fortunately, it is possible to build more complex primitives from a combination of these existing resources. Since the **atomic** combinator may be nested without restriction, complex operations on collections of shared state objects can be made atomic within a **`def`** or other abstraction. The only disadvantage of this approach is the increase in conflicts, since the Ora semantics can only merge individual resources to resolve conflicts; it cannot express the more complex properties of coordinated sets of resources.

What other resources might be included as primitives in future extensions of Ora? One critically important capability that is currently absent from Ora is a primitive `Set` site, to express a mutable set with addition, removal, membership, and so on. It is possible to express a set using a `Ref` containing some standard set representation, but this causes spurious conflicts, when for example a child transaction adds a new element, and a parent removes a different element. With a primitive `Set`, these operations would not conflict, but using just a `Ref`, they would be divergent writes to the underlying list. Even with a more sophisticated linked structure using many `Ref` instances, there is still a potential for false conflict. Expanding the formal semantics to support a `Set` site would allow Ora to compare more favorably with a system like Galois, which is designed to heavily exploit ”don't care” nondeterminism [PNK$^+$11]. Currently, Ora only weakly exploits this form of nondeterminism with channel and semaphore operations.

9.2 Incorporating Time

The Ora semantics does not incorporate time. In particular, the site `Rwait` is not considered part of the theory; as explained in Section 6.1.1, it does not interoperate with transactions at all, and so the properties of atomicity and coatomicity might not hold in programs that make use of `Rwait` to order events temporally, rather than causally.

Real time is excluded from the theory partly because the timed semantics of Orc is not nearly as well-established as its untimed semantics. The operational semantics of Ora are based on the untimed operational semantics of Orc, since the untimed theory is more developed. In the untimed semantics, individual transitions of the program and of sites have no time information associated with them, so `Rwait` is practically incapable of providing any guarantees about the ordering of events; a program transition could be delayed by an arbitrary amount of real time.

If the semantics of Ora did take account of event timing, then `Rwait` could be used to temporally order events. This would cause significant problems with the maintenance of atomicity and coatomicity, since every event would now have, as implicit causes, any events that occurred at preceding times. In order for atomicity and coatomicity to hold, the algorithm must take account of the causalities induced by temporal ordering.

The issues that arise in this situation actually closely parallel the issues that prompted the logical versioning algorithm. Logical versions are passed back and forth between the Ora program and the environment, so that sites can correctly account for the causalities induced by the program. In order to account for the causalities induced by time, an extension of that versioning system may suffice. Each logical version becomes a tuple of a natural number, which is used as before, and a timestamp. Whenever a call to `Rwait` responds, it responds with a logical version that has its timestamp set to the current time.[1] Whenever a transaction begins, its initial version gets the current time as its timestamp. Logical versions would then be partially ordered, but this would not cause any problems. The filter operation $[< n]$ used in the snapshot judgment would become $[< (n, t)]$, filtering out any events with logical version n or greater, or with timestamp t or greater. The commit version would be calculated as the least upper bound of each participant version, together with the initial version, which is simply the max of each n and each t.

This technique has an obvious analogue in the distributed systems literature:

[1] This includes calls of the form `Rwait(0)`, which would provide an interesting function for that otherwise unused case: such a call would ensure that the caller's timestamp is advanced to the current time.

vector clocks [Fid88]. In this case, rather than expressing ordering of operations on individual machines, we are expressing ordering of operations with respect to different subsets of the causality relation: some are induced by orchestration of the program and the environment, while others are induced by observations of the realtime clock.

Using this same technique, we could expand such a vector clock to add a third element: virtual time. Virtual time has been explored as a way of orchestrating simulations in Orc [KPM08], and it could be managed in a way similar to real time as described above. However, the current algorithm for virtual time uses a program property called "quiescence" to determine when to advance the virtual clock, and it is not clear how quiescence would interact with the **atomic** combinator.

9.3 Blocking

Many operations on transactional resources in Ora are allowed to block. Specifically, a semaphore `acquire`, a channel `get`, or a reference/cell `read` might block for some indefinite period before returning a value. The call could be waiting for the semaphore to become available, for the channel to contain an available item, or for the reference to be written.

In each case, the call can only unblock if some other operation occurs on the shared resource. Furthermore, in order for the call to unblock, the operation that occurs must be visible to the blocked call, according to the rules of atomicity and coatomicity. In particular, it is possible for the observed state of a resource to be unavailable, empty, or unwritten, so that a call within a transaction could block indefinitely while all operations outside the transaction which would unblock the resource are hidden from the transaction due to coatomicity. This is represented concretely in the observed state algorithm in Section 6.2; if a call blocks, and no ancestor node contains that call in its waiting set (because it was purged or simply never added), then no external operation can unblock that call.

This possibility of indefinite blocking may be the reason why blocking operations have not usually been incorporated into most transactional systems. However, those systems are typically operating under a strong limitation: no concurrent activity is allowed within a transaction. So, allowing a call to block within a transaction would clearly be a poor choice, since the entire transaction would block forever.

Ora, on the other hand, does allows concurrency within transactions; the body expression of an **atomic** is an arbitrary Orc expression, and as such it could contain any number of parallel activities. Ora's versioning algorithms are explicitly designed to accommodate this possibility. The ability to evaluate concurrently within a transaction is especially important in this case, since allowing a blocking call within a transaction becomes sensible if concurrent activity *from within that same transaction* could unblock the call.

Nevertheless, programmers must use caution when performing resource operations that may block, if the expected unblocking operation might not come from within the same transaction. If the intended unblocking operation is in a descendant transaction, it is possible that transaction might abort, and never successfully commit the necessary resource change. If the unblocking operation is in an ancestor transaction, the versioning algorithm may prevent it from being seen; even if the operation could theoretically be seen and still preserve coatomicity, the algorithm might fail to do so, since it is by necessity a conservative approximation. If the unblocking operation is in an unrelated transaction, the problem is compounded: if *any* operation in that transaction is not visible, due to coatomicity, then *all* of its operations must be hidden, due to atomicity.

There are various ways that a programmer might address this problem. Simply being aware of its existence, and writing programs accordingly, may suffice in many cases. If this is not feasible, then an expression prone to blocking on external operations could be augmented with a timeout, using the atomic timeout technique shown in Section 5.3.4.

A small extension to Ora itself may also help: semaphores, channels, and cells could be augmented with alternate operations that halt, rather than blocking, if the operation cannot complete. The Orc standard library sites already allow such operations, such as the channel operation `getD`, which acts like `get`, except that it halts if the channel is empty. Adding these operations would only be a minor extension of Ora's site semantics, with no changes needed in the underlying algorithm.

A more intricate extension is also possible: alternate operations that halt rather than blocking, but only if the operation is not in the waiting set for any ancestor. Call these methods `getX`, `acquireX`, and `readX`, for the channel, semaphore, and cell cases respectively. The use of these methods expresses the expectation that the operation should be unblocked by an outside operation, and once that becomes impossible, this should be indicated by halting. This neatly solves many of the problems that arise with blocking on ancestor operations. For example, here is a repaired version of the Dining Philosophers program from Section 5.3.3:

```
def philosopher(l,r) =
  think() >>
  atomic ( (l.acquireX() & r.acquireX()); Abort() ) >>
  eat() >>
  atomic (  l.release() & r.release() ) >>
  philosopher(l,r)
```

Now, whenever an `acquire` becomes blocked indefinitely by becoming stuck in a snapshot that cannot see new `release` operations, the `acquireX` method will halt, causing the `Abort` site to be called, aborting the transaction. Due to the use of unary **`atomic`**, the transaction will immediately be retried. However, the initial version of the retry will now be greater than all logical versions at the parent resource node, so the transaction will not abort again in the same fashion.

9.4 Nonserializable Executions

Extending Orc with the **atomic** combinator increases the expressive power of the language, and not just in its capability to support transactions. Its inclusion has another surprising consequence. Consider the following program:

```
val r = Ref(0)
val s = Ref(0)

val a = atomic (s := 1 >> r?)
val b = atomic (r := 1 >> s?)

(a,b)
```

In the classic model of concurrent program reasoning, we would consider the possible sequences of events in the execution of this program, such that the executions of the **atomic** expressions occurred in some serial order. For example, the first **atomic** expression could execute followed by the second, resulting in `(0,1)`, or the second could execute and then the first, resulting in `(1,0)`.

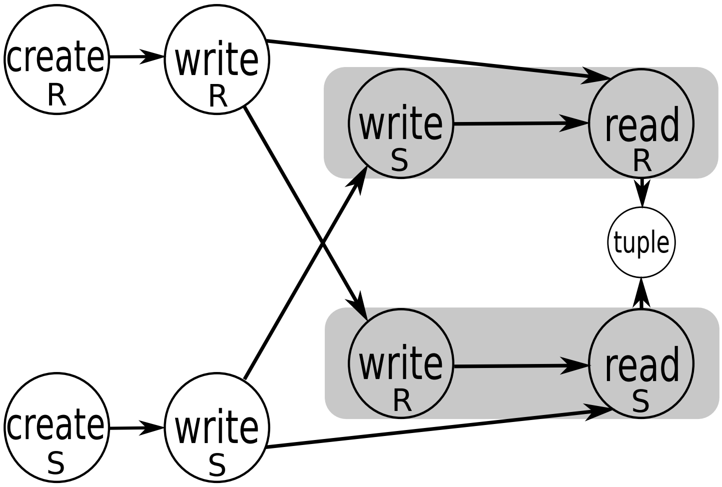


Figure 9.1: No equivalent serial order

However, in Ora, those are not the only possible behaviors; there is a third possibility, resulting in `(0,0)`. The event graph corresponding to that possibility is shown in Figure 9.1.

To those accustomed to sequential programming, this may seem like a surprising and counterintuitive result; the execution that publishes `(0,0)` has *no* equivalent serial ordering![2] More precisely, it violates *sequential consistency*, as defined by Lamport [Lam79]. However, the absence of serial ordering is actually a better fit for many modern processor architectures, since it admits weaker memory models and thus enables many optimizations. A programming technique called *relativistic programming* has arisen to specifically take advantage of these situations [HW12], and each of the arguments made in favor of relativistic programming apply equally to Ora.

In a previous attempt to formalize Ora, the consistency property stated that for any execution that used transactions, there existed an "erased" execution with equivalent behavior but no use of transactions. In other words, the **atomic** combinator only constrained program behavior; it did not enable behaviors that were previously inexpressible. In fact, such a consistency property cannot possibly be true given the current semantics of Orc and Ora, because the **atomic** combinator

[2] Technically this is only true under the assumption that a read in a trace sees the most recent write in the trace, but weakening that condition is arguably just as surprising.

actually *does* expand the expressive power of Orc, by providing the capability for nonserializable execution behaviors, as we have just seen. This is because the operational semantics of Orc enforces sequential consistency, but this enforcement is circumvented by partitioning effects on shared state into separate parallel transactions as Ora does.

In the future, with a more sophisticated operational semantics (or perhaps a denotational semantics based on partial orders), Orc itself could have the capability to produce nonserializable executions, and then the **atomic** combinator would not introduce new behaviors, reopening the possibility of formalizing consistency of an execution in terms of an equivalence to another execution with all transactions erased.

Bibliography

[ABZ07] Lucia Acciai, Michele Boreale, and Silvano Dal Zilio. A concurrent calculus with atomic transactions. In *Proceedings of the 16th European conference on Programming*, ESOP'07, pages 48–63, Berlin, Heidelberg, 2007. Springer-Verlag.

[AFS08] Kunal Agrawal, Jeremy T. Fineman, and Jim Sukha. Nested parallelism in transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 163–174, New York, NY, USA, 2008. ACM.

[Arm07] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[BCF04] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, September 2004.

[BG83] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control–theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, December 1983.

[BN09] P.A. Bernstein and E. Newcomer. *Principles of Transaction Processing*, chapter 8. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 2009.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[Com13] Cω. `http://research.microsoft.com/comega/`, July 2013.

[CRS06] João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, December 2006.

[Dij65] Edsger W. Dijkstra. Cooperating sequential processes (ewd-123), 1965.

[Dij68] Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, March 1968.

[Dij83] Edsger W. Dijkstra. The distributed snapshot of chandy/lamport/misra (ewd-864), 1983.

[EDKG08] Laura Effinger-Dean, Matthew Kehrt, and Dan Grossman. Transactional events for ML. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 103–114, New York, NY, USA, 2008. ACM.

[FG02] Cédric Fournet and Georges Gonthier. The join calculus: A language for distributed mobile programming. In *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, pages 268–332, London, UK, UK, 2002. Springer-Verlag.

[Fid88] Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proc. of the 11th Australian Computer Science Conference (ACSC'88)*, pages pp 56–66, February 1988.

[FM06] Azadeh Farzan and P. Madhusudan. Causal atomicity. In *Proceedings of the 18th international conference on Computer Aided Verification*, CAV'06, pages 315–328, Berlin, Heidelberg, 2006. Springer-Verlag.

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[Gro07] Dan Grossman. The transactional memory / garbage collection analogy. In *Proceedings of the 22nd annual ACM SIGPLAN conference on*

Object-oriented programming systems and applications, OOPSLA '07, pages 695–706, New York, NY, USA, 2007. ACM.

[Hew10] Carl Hewitt. Actor model for discretionary, adaptive concurrency. *CoRR*, abs/1008.1459, 2010.

[HF03] Tim Harris and Keir Fraser. Language support for lightweight transactions. *SIGPLAN Not.*, 38(11):388–402, October 2003.

[HM93] Maurice Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

[HMJH05] Tim Harris, Simon Marlow, Simon P. Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[HW12] Philip W. Howard and Jonathan Walpole. A case for relativistic programming. In *Proceedings of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability*, RACES '12, pages 33–38, New York, NY, USA, 2012. ACM.

[Jav13] The Java tutorials: Intrinsic locks and synchronization. `http://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html`, July 2013.

[KCM06] David Kitchin, William R. Cook, and Jayadev Misra. A language for task orchestration and its semantic properties. In Christel Baier and Holger Hermanns, editors, *CONCUR 2006 – Concurrency Theory*, volume 4137 of *Lecture Notes in Computer Science*, pages 477–491. Springer, 2006.

[KPM08] David Kitchin, Evan Powell, and Jayadev Misra. Simulation using orchestration (extended abstract). In José Meseguer and Grigore Roşu, editors, *Algebraic Methodology and Software Technology*, volume 5140 of *Lecture Notes in Computer Science*, pages 2–15. Springer, 2008.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[Lam13] Simon S. Lam. Binary exponential backoff in Ethernet: Origin. `http://www.cs.utexas.edu/users/lam/NRL/backoff.html`, July 2013.

[LR80] Butler W. Lampson and David D. Redell. Experience with processes and monitors in Mesa. *Commun. ACM*, 23(2):112–113, February 1980.

[MBL06] Milo Martin, Colin Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2), 2006.

[Mil99] Robin Milner. *Communicating and mobile systems: the π-calculus.* Cambridge University Press, New York, NY, USA, 1999.

[Orc13a] Orc reference manual. `http://orc.csres.utexas.edu/documentation/html/refmanual/index.html`, July 2013.

[Orc13b] Orc reference manual: EBNF grammar. `http://orc.csres.utexas.edu/documentation/html/refmanual/ref.syntax.EBNF.html`, July 2013.

[Orc13c] Orc reference manual, standard library: state. `http://orc.csres.utexas.edu/documentation/html/refmanual/ref.stdlib.state.html`, July 2013.

[PNK$^+$11] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 12–25, New York, NY, USA, 2011. ACM.

[PT00] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000.

[Rep99] John H. Reppy. *Concurrent programming in ML.* Cambridge University Press, New York, NY, USA, 1999.

[RG05] Michael F. Ringenburg and Dan Grossman. AtomCaml: first-class atomicity via rollback. *SIGPLAN Not.*, 40(9):92–104, September 2005.

[RHW10] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 47–56, New York, NY, USA, 2010. ACM.

[RW09] Hany E. Ramadan and Emmett Witchel. The Xfork in the road to coordinated sibling transactions. *TRANSACT*, February 2009.

[SQL92] Information Technology - Database Language - SQL. *International Standard ISO/IEC 9075*, 1992.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[VWAT$^+$09] Haris Volos, Adam Welc, Ali-Reza Adl-Tabatabai, Tatiana Shpeisman, Xinmin Tian, and Ravi Narayanaswamy. NePaLTM: Design and implementation of nested parallelism for transactional memory systems. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 123–147, Berlin, Heidelberg, 2009. Springer-Verlag.

[Win89] Glynn Winskel. An introduction to event structures. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop*, pages 364–397, London, UK, UK, 1989. Springer-Verlag.